Intro to Programming in Python

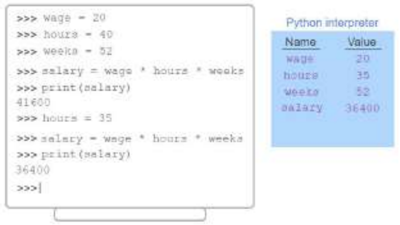
Programming using Python

The **Python interpreter** is a computer program that executes code written in the Python programming language

An **interactive interpreter** is a program that allows the user to execute one line of code at a time

The interactive interpreter displays a **prompt** ("&gt;&gt;&gt;") that indicates the interpreter is ready to accept code

A programmer can write Python code in a file, and then provide that file to the interpreter. The interpreter begins by executing the first line of code at the top of the file, and continues until the end is reached



```
>>> wage = 20
>>> hours = 40
>>> weeks = 52
>>> salary = wage * hours * weeks
>>> print(salary)
41600
>>> hours = 35
>>> salary = wage * hours * weeks
>>> print(salary)
36400
>>>
```

Python interpreter

Name	Value
wage	20
hours	35
weeks	52
salary	36400

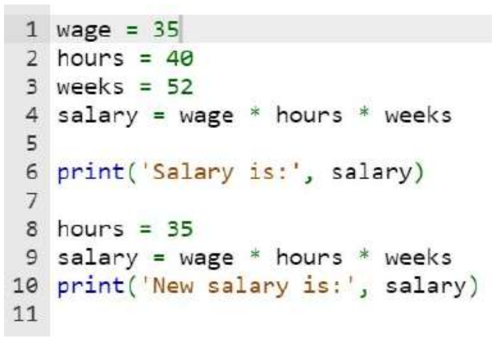
Users can change values and execute calculations again.

Captions ^

1. After each press of the enter key, the python interpreter executes the line of code.
2. The python interpreter can be used as a calculator and can perform a variety of calculations.
3. Users can change values and execute calculations again.

- The **print()** function displays variables or expression values

- The **#** character denote **comments**



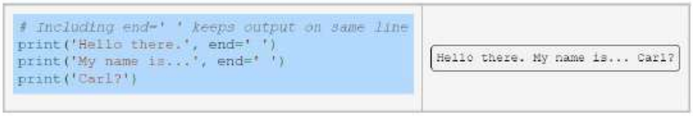
```
wage = 35
hours = 40
weeks = 52
salary = wage * hours * weeks

print('Salary is:', salary)

hours = 35
salary = wage * hours * weeks
print('New salary is:', salary)
```

Text enclosed in quotes is known as a **string literal**

-A string literal can be surrounded by matching single or double quotes

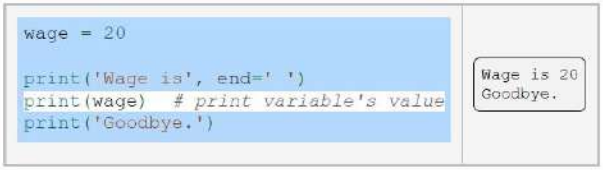
Each print() starts a new line

A programmer can add end=' ') inside of print() to keep the output of a subsequent print statement on the same line separated by a single space



```
# Including end=' ' keeps output on same line
print('Hello there.', end=' ')
print('My name is...', end=' ')
print('Carl?')
```

```
Hello there. My name is... Carl?
```



```
wage = 20

print('Wage is', end=' ')
print(wage)    # print variable's value
print('Goodbye.')
```

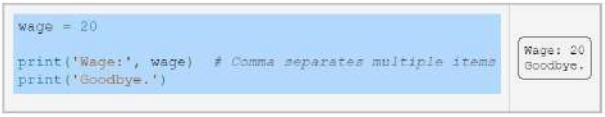
```
Wage is 20
Goodbye.
```

The value of a variable can be printed out via: print (variable_name) with no quotation marks

Outputting multiple items with one statement

Programmers commonly try to use a single print statement for each line of output by combining the printing of text, variable values, and new lines. A programmer can simply separate the items with commas, and each item in the output will be separated by a space. Combining string literals, variables, and new lines can improve program readability, because the program's code corresponds more closely to the program's printed output.

Figure 2.3.4: Printing multiple items using a single print statement.



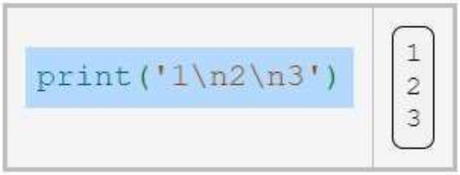
```
wage = 20

print('Wage:', wage)  # Comma separates multiple items
print('Goodbye.')
```

```
Wage: 20
Goodbye.
```

Need help? Feedback?

A common error is to forget the comma between items, as in `print('Name' user_name)`.

Output can be moved to the next line by printing "\n", known as a newline character

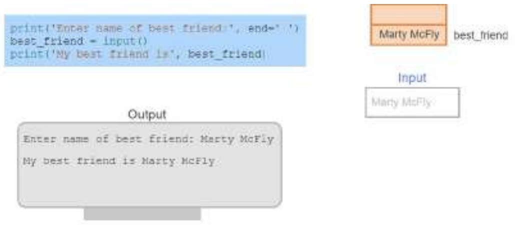


```
print('1\n2\n3')
```

```
1
2
3
```

print() always adds a newline character after the output automatically to move the next output to the next row, unless end=' ` is provided

Any space, tab, or newline is called whitespace



```
print('Enter name of best friend:', end=' ')
best_friend = input()
print('My best friend is', best_friend)
```

```
Enter name of best friend: Marty McFly
My best friend is Marty McFly
```

best_friend's value can then be used in subsequent processing and outputs.

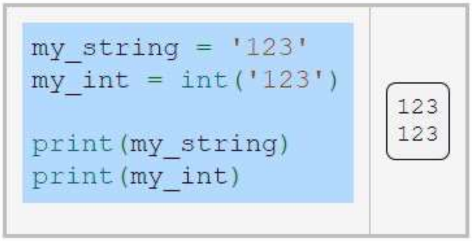
The statement best_friend = input() will read text entered by the user and the best_friend variable is assigned with the entered text

Captions

1. The input() statement gets an input value from the keyboard and puts that value into the best_friend variable.
2. best_friend's value can then be used in subsequent processing and outputs.

The input obtained by input() is any text that a user typed, including numbers, letters, or special characters. Such text in a computer program is called a string

The '123' string is a sequence of characters whereas 123 represents the integer value - each are an example of a type; integers can be divided by 2, but not strings

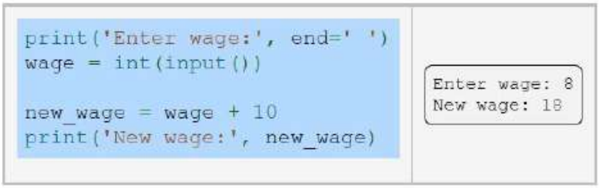


```
my_string = '123'
my_int = int('123')

print(my_string)
print(my_int)
```

```
123
123
```

Reading from input always results in a string type. If a string contains numbers, like '123', then the int() function can be used to convert that string to the integer 123

A combined input() and int() can read a string from the user and then convert that string to an integer for use in a calculation



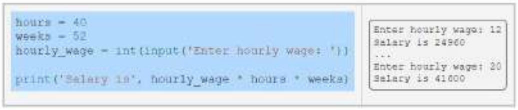
```
print('Enter wage:', end=' ')
wage = int(input())

new_wage = wage + 10
print('New wage:', new_wage)
```

```
Enter wage: 8
New wage: 18
```

Input prompt

Adding a string inside the parentheses of input() displays a prompt to the user before waiting for input and is a useful shortcut to adding an additional print statement line.

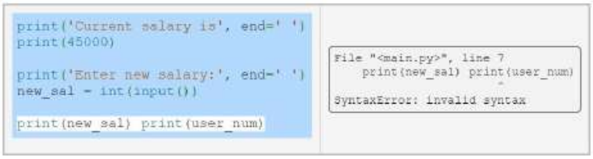
Figure 2.3.9: Basic input example.



```
hours = 40
weeks = 52
hourly_wage = int(input('Enter hourly wage: '))

print('Salary is', hourly_wage * hours * weeks)
```

```
Enter hourly wage: 12
Salary is 24960
...
Enter hourly wage: 20
Salary is 41600
```

Errors



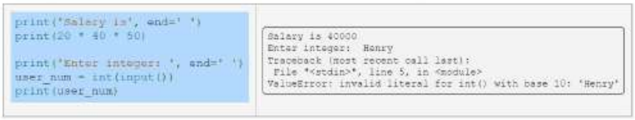
```
print('Current salary is', end=' ')
print(45000)

print('Enter new salary:', end=' ')
new_sal = int(input())

print(new_sal) print(user_num)
```

```
File "<main.py>", line 7
    print(new_sal) print(user_num)

SyntaxError: invalid syntax
```

Syntax error violates a programming language's rules on how symbols can be combined to create a program

Runtime error wherein a program's syntax is correct but the program attempts an impossible operation, such as dividing by zero

The program crashes because the user enters 'Henry' instead of an integer value.

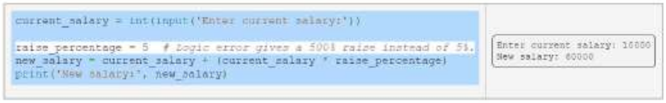


```
print('Salary is', end=' ')
print(20 * 40 * 50)

print('Enter integer: ', end=' ')
user_num = int(input())
print(user_num)
```

```
Salary is 40000
Enter integer:  Henry
Traceback (most recent call last):
 File "<stdin>", line 5, in <module>
ValueError: invalid literal for int() with base 10: 'Henry'
```

Crash an abrupt and unintended termination of a program

The below program attempts to calculate a 5% raise for an employee's salary. The programmer made a mistake by assigning raise_percentage with 5, instead of 0.05, thus giving a happy employee a 500% raise.

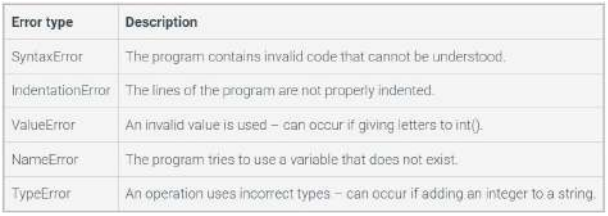


```
current_salary = int(input('Enter current salary:'))

raise_percentage = 5  # Logic error gives a 500% raise instead of 5%.
new_salary = current_salary + (current_salary * raise_percentage)
print('New salary:', new_salary)
```

```
Enter current salary: 10000
New salary: 60000
```

Logic error subtle enough not to cause an runtime error and a crash - often called a **bug**

Code development is usually done with an **integrated development environment (IDE)**



Error type	Description
SyntaxError	The program contains invalid code that cannot be understood.
IndentationError	The lines of the program are not properly indented.
ValueError	An invalid value is used – can occur if giving letters to int().
NameError	The program tries to use a variable that does not exist.
TypeError	An operation uses incorrect types – can occur if adding an integer to a string.

Instructions represented as 0s and 1s are known as **machine instructions**, and a sequence of machine instructions together form an **executable program**

Programmers created programs called assemblers to automatically translate human readable instructions known as **assembly** language instructions, into machine instructions high-level languages created to support programming using formulas or algorithms

- Compilers programs that automatically translate high-level language programs into executable programs

transistors are integrated onto a single chip called an integrated circuit (IC)

- Moore's Law the doubling of IC capacity roughly every 18 months

Programmers began creating scripting languages to execute programs without the need for compilation. A script is executed by another program called an interpreter

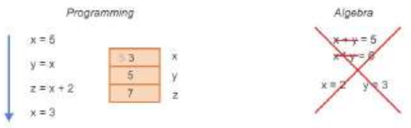
- -Python is a scripting language

Variables and Assignments

In a program, a variable is a named item, such as x or numPeople, used to hold a value

An assignment statement assigns a variable with a value, such as x=5

- x keeps that value during subsequent statements, until x is assigned again



The = isn't "equals", but is an action that PUTS a value into the variable. Assignment statements only make sense when executed in sequence.

Captions

1. In programming, a variable is a place to hold a value. Here, variables x, y, and z are depicted graphically as boxes.
2. An assignment statement assigns the left-side variable with the right-side expression's value. x = 5 assigns x with 5.
3. y = x assigns y with x's value, which presently is 5. z = x + 2 assigns z with x's present value plus 2, so 5 + 2 or 7.
4. A subsequent x = 3 statement assigns x with 3. x's former value of 5 is overwritten and thus lost. Note that the values held in y and z are unaffected, remaining as 5 and 7.
5. In algebra, an equation means "the item on the left always equals the item on the right". So for x + y = 5 and x * y = 6, one can determine x = 2 and y = 3 or vice versa.
6. Assignment statements look similar to algebra equations but have VERY different meaning. The left side MUST be one variable.
7. The = isn't "equals", but is an action that PUTS a value into the variable. Assignment statements only make sense when executed in sequence.

In programming, = is not equals

Valid	Invalid
x=1	x+1=3
x=y	x+y=y+x
x=y+2	

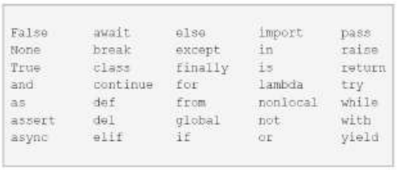
A variable may appear on both the left and right side of a statement, as in x=x+1. Increasing a variable's value by 1 is common and known as **incrementing** the variable

Identifiers

A name created by a programmer for an item like a variable or function is called an **identifier**

- be a sequence of letters, underscores, and digits
- start with a letter or an underscore

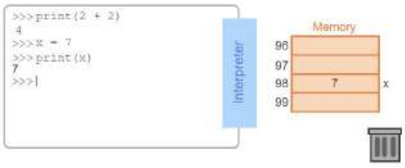
Identifiers are case sensitive. A **reserved word** is a word that is part of the language, like integer, Get, or Put that can't be used as an identifier



False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Objects

An **object** represents a value and is automatically created by the interpreter when executing a line of code



```
>>>print(2 + 2)
4
>>>x = 7
>>>print(x)
7
>>>|
```

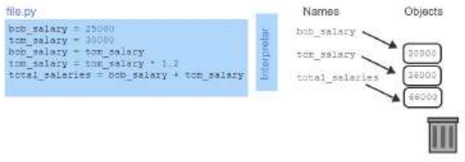
Objects are retrieved from memory when needed.

Deleting unused objects is an automatic process called **garbage collection** that helps keep the memory of the computer less utilized

Captions

1. The interpreter creates a new object with the value 4. The object is stored somewhere in memory.
2. Once 4 is printed, the object is no longer needed and is thrown away.
3. New object created: 'x' references object stored in address 98.
4. Objects are retrieved from memory when needed.

Name binding is the process of associating names with interpreter objects



```
bob_salary = 25000
tom_salary = 30000
bob_salary = tom_salary
tom_salary = tom_salary * 1.2
total_salaries = bob_salary + tom_salary
```

total_salaries object is created by the interpreter and is assigned bob_salary + tom_salary

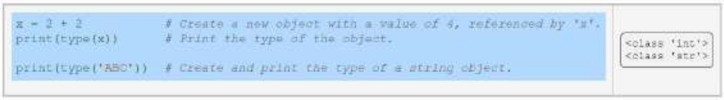
Each Python object has three defining properties:

Captions

1. bob_salary object is created by the interpreter.
2. tom_salary object is created by the interpreter.
3. bob_salary is assigned tom_salary, and the 25000 object is garbage collected.
4. tom_salary is assigned tom_salary * 1.2
5. total_salaries object is created by the interpreter and is assigned bob_salary + tom_salary

- **Value:** Such as "20", "abcdef", or 55.
- **Type:** such as integer, float, or string
- **Identity:** a unique identifier that describes the object

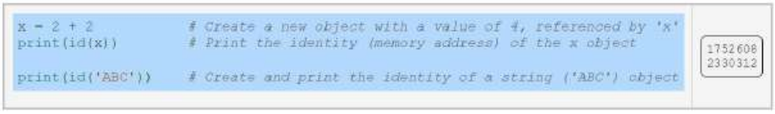
An object's type never changes once created. The built-in function **type()** returns the type of an object



```
x = 2 + 2              # Create a new object with a value of 4, referenced by 'x'.
print(type(x))         # Print the type of the object.

print(type('ABC'))     # Create and print the type of a string object.
```

```
<class 'int'>
<class 'str'>
```

The type of an object also determines the mutability of an object. **Mutability** indicates whether the object's value is allowed to be changed. Integers and strings are **immutable**



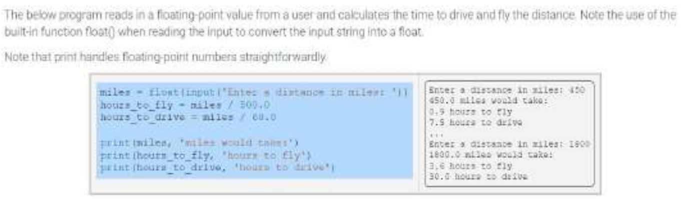
```
x = 2 + 2              # Create a new object with a value of 4, referenced by 'x'
print(id(x))           # Print the identity (memory address) of the x object

print(id('ABC'))       # Create and print the identity of a string ('ABC') object
```

```
1752608
2330312
```

The built-in function **id()** gives the value of an object's identity

Numeric types: Floating-point

A **floating-point number** is a real number, like 48.6, 0.0001, or -444.45. The term "floating-point" refers to the decimal point being able to appear anywhere ("float") in the number. A variable declared as type **float** stores a floating-point number


The below program reads in a floating-point value from a user and calculates the time to drive and fly the distance. Note the use of the built-in function float() when reading the input to convert the input string into a float.

Note that print handles floating-point numbers straightforwardly.

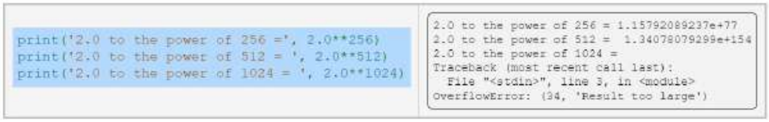
```python
miles = float(input('Enter a distance in miles: '))
hours_to_fly = miles / 500.0
hours_to_drive = miles / 60.0

print(miles, 'miles would take:')
print(hours_to_fly, 'hours to fly')
print(hours_to_drive, 'hours to drive')
```

```
Enter a distance in miles: 450
450.0 miles would take:
0.9 hours to fly
7.5 hours to drive
...
Enter a distance in miles: 1800
1800.0 miles would take:
3.6 hours to fly
30.0 hours to drive
```

A **floating-point literal** is a number with a fractional part, even if that fraction is 0, as in 1.0, 0.0, or 99.573

A floating-point literal using **scientific notation** is written using an e preceding the power-of-10 exponent, as in 6.02e23 to represent 6.02×10^{23}

Float-type objects have a limited range from 1.8×10^{308} to 2.3×10^{-308}. Assigning a floating-point value outside of this range generates an **OverFlow Error**



```python
print('2.0 to the power of 256 =', 2.0**256)
print('2.0 to the power of 512 = ', 2.0**512)
print('2.0 to the power of 1024 = ', 2.0**1024)
```

```
2.0 to the power of 256 = 1.15792089237e+77
2.0 to the power of 512 =  1.34078079299e+154
2.0 to the power of 1024 =
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
OverflowError: (34, 'Result too large')
```

The syntax for outputting the float myFloat with two digits after the decimal point is

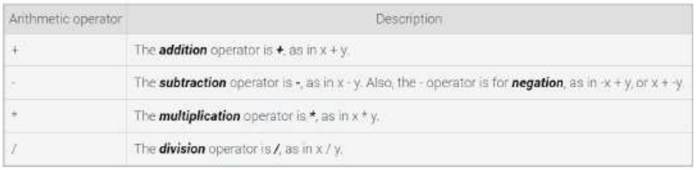
```python
print(f'{myFloat:.2f}')
```

Arithmetic expressions

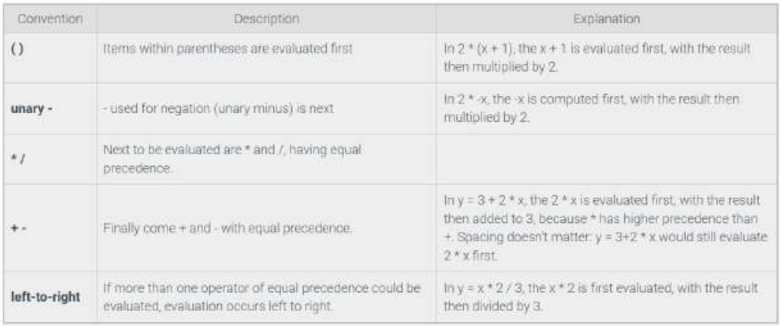
An **expression** is a combination of items, like variables, literals, operators, and parenthesis, that evaluates to a value, like 2*(x+1)

A literal is a specific value in code like 2

An operator is a symbol that performs a built-in calculation



Arithmetic operator	Description
+	The ***addition*** operator is **+**, as in x + y.
-	The ***subtraction*** operator is **-**, as in x - y. Also, the - operator is for ***negation***, as in -x + y, or x + -y.
*	The ***multiplication*** operator is *****, as in x * y.
/	The ***division*** operator is **/**, as in x / y.



Convention	Description	Explanation
()	Items within parentheses are evaluated first	In 2 * (x + 1), the x + 1 is evaluated first, with the result then multiplied by 2.
unary -	- used for negation (unary minus) is next	In 2 * -x, the -x is computed first, with the result then multiplied by 2.
* /	Next to be evaluated are * and /, having equal precedence.	
+ -	Finally come + and - with equal precedence.	In y = 3 + 2 * x, the 2 * x is evaluated first, with the result then added to 3, because * has higher precedence than +. Spacing doesn't matter: y = 3+2 * x would still evaluate 2 * x first.
left-to-right	If more than one operator of equal precedence could be evaluated, evaluation occurs left to right.	In y = x * 2 / 3, the x * 2 is first evaluated, with the result then divided by 3.

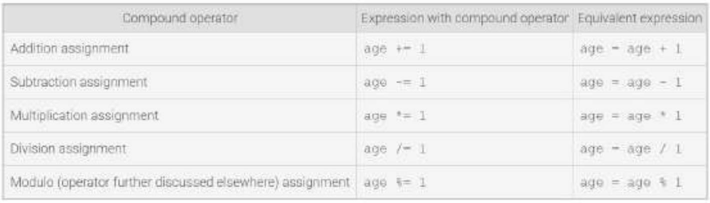
An expression evaluates to a value, which replaces the expression

An expression is evaluated in programming using these precedence rules

Python expressions

Minus (-) used as a negative is known as unary minus

Special operators called compound operators provide a shorthand way to update a variable, such as +=1 being shorthand for age = age + 1



Compound operator	Expression with compound operator	Equivalent expression
Addition assignment	`age += 1`	`age = age + 1`
Subtraction assignment	`age -= 1`	`age = age - 1`
Multiplication assignment	`age *= 1`	`age = age * 1`
Division assignment	`age /= 1`	`age = age / 1`
Modulo (operator further discussed elsewhere) assignment	`age %= 1`	`age = age % 1`

Division and Modulo

Division: Integer rounding

The division operator / performs division and returns a floating-point number. Ex:

- 20 / 10 is 2.0.
- 50 / 50 is 1.0.
- 5 / 10 is 0.5.

The floor division operator // can be used to round down the result of a floating-point division to the closest smaller whole number value. The resulting value is an integer type if both operands are integers; if either operand is a float, then a float is returned.

- 20 // 10 is 2.
- 50 // 50 is 1.
- 5 // 10 is 0. (5/10 is 0 and the remainder 5 is thrown away).
- 5.0 // 2 is 2.0

For division, the second operand of / or // must never be 0, because division by 0 is mathematically undefined.

The **modulo operator (%)** evaluates to the remainder of the division of two integer operands

24 % 10 is 4. Reason: 24/10 is 2 with remainder 4

1 % 2 is 1. Reason: 1/2 is 0 with remainder 1

Given a number, % and // can be used to get each digit. For a 3-digit number user_val like 927:

```
ones_digit      = user_val % 10     # Ex: 927 % 10 is 7.
tmp_val         = user_val // 10

tens_digit      = tmp_val % 10      # Ex: tmp_val = 927 // 10 is 92. Then 92 % 10 is 2.
tmp_val         = tmp_val // 10

hundreds_digit = tmp_val % 10      # Ex: tmp_val = 92 // 10 = 9. Then 9 % 10 is 9.
```

Given a 10-digit phone number stored as an integer, % and // can be used to get any part, such as the prefix. For phone_num = 9365551212 (whose prefix is 555):

```
tmp_val = phone_num // 10000  # // 10000 shifts right by 4, so 936555.
prefix_num = tmp_val % 1000 # % 1000 gets the right 3 digits, so 555.
```

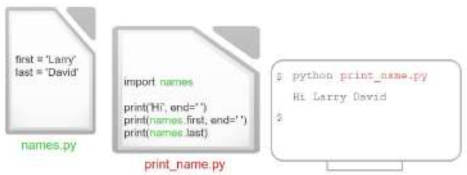
Dividing by a power of 10 shifts a value right. Ex: 321 // 10 is 32. Ex: 321 // 100 is 3.

% by a power of 10 gets the rightmost digits. Ex: 321 % 10 is 1. Ex: 321 % 100 is 21.

Module Basics

A **module** is a file containing Python code that can be used by other modules or scripts

A module is made available for use via the **import** statement. Once a module is imported, any object defined in that module can be accessed using **dot notation**



names.py:
```
first = 'Larry'
last = 'David'
```

print_name.py:
```
import names

print('Hi', end=' ')
print(names.first, end=' ')
print(names.last)
```

```
$ python print_name.py
  Hi Larry David
$
```

Running the script imports the module and accesses the module contents using dot notation.

Captions ^

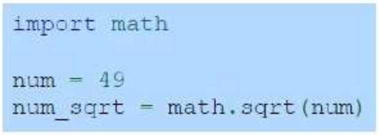
1. Code can be separated into multiple files. The names.py module has some predefined variables.
2. The print_name.py script imports variables from names.py using dot notation.
3. Running the script imports the module and accesses the module contents using dot notation.

When a module is imported, all code in the module is immediately executed. A built-in special name __name__ is used to determine if the file was executed as a script by the programmer, or if the file was imported by another module

If the value of __name__ is the string '__main__', then the file was executed as a script

Math Module

Python comes with a standard math module to support more advanced math operations



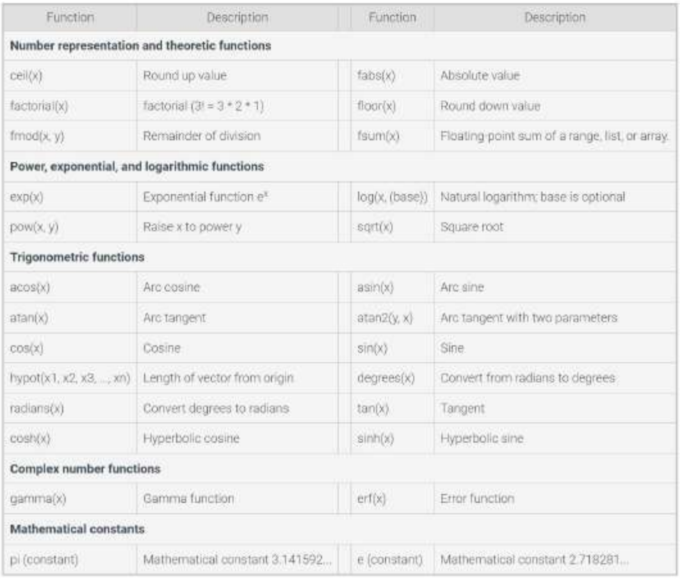
```
import math

num = 49
num_sqrt = math.sqrt(num)
```

sqrt() is known as a function

A function is a list of statements executed by referring to the function's name

The process of invoking a function is referred to as a function call. The item passed to a function is referred to as an argument

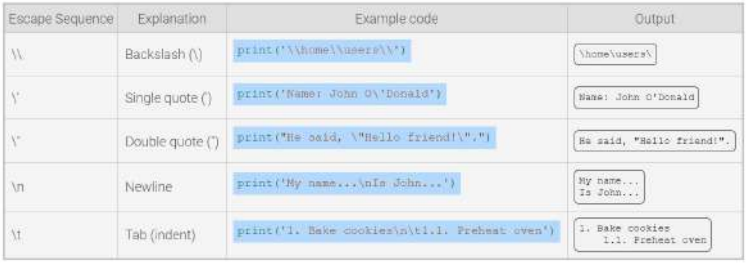
Table 3.9.1: Functions in the standard math module.



Function	Description	Function	Description
Number representation and theoretic functions			
ceil(x)	Round up value	fabs(x)	Absolute value
factorial(x)	factorial (3! = 3 * 2 * 1)	floor(x)	Round down value
fmod(x, y)	Remainder of division	fsum(x)	Floating-point sum of a range, list, or array.
Power, exponential, and logarithmic functions			
exp(x)	Exponential function e^x	log(x, (base))	Natural logarithm; base is optional
pow(x, y)	Raise x to power y	sqrt(x)	Square root
Trigonometric functions			
acos(x)	Arc cosine	asin(x)	Arc sine
atan(x)	Arc tangent	atan2(y, x)	Arc tangent with two parameters
cos(x)	Cosine	sin(x)	Sine
hypot(x1, x2, x3, ..., xn)	Length of vector from origin	degrees(x)	Convert from radians to degrees
radians(x)	Convert degrees to radians	tan(x)	Tangent
cosh(x)	Hyperbolic cosine	sinh(x)	Hyperbolic sine
Complex number functions			
gamma(x)	Gamma function	erf(x)	Error function
Mathematical constants			
pi (constant)	Mathematical constant 3.141592...	e (constant)	Mathematical constant 2.718281...

Representing Text

Python uses Unicode to represent every possible character as a unique number, known as a code point



Escape Sequence	Explanation	Example code	Output
\\	Backslash (\)	`print('\\home\\users\\')`	`\home\users\`
\'	Single quote (')	`print('Name: John O\'Donald')`	`Name: John O'Donald`
\"	Double quote (")	`print("He said, \"Hello friend!\".")`	`He said, "Hello friend!".`

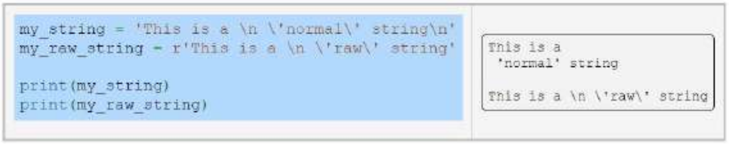
\n	Newline	`print('My name...\nIs John...')`	`My name...`
			`Is John...`

\t	Tab (indent)	`print('1. Bake cookies\n\t1.1. Preheat oven')`	`1. Bake cookies`
			`    1.1. Preheat oven`

A newline character is encoded as 10 and uses the two-item sequence \n

The \ is known as a backslash

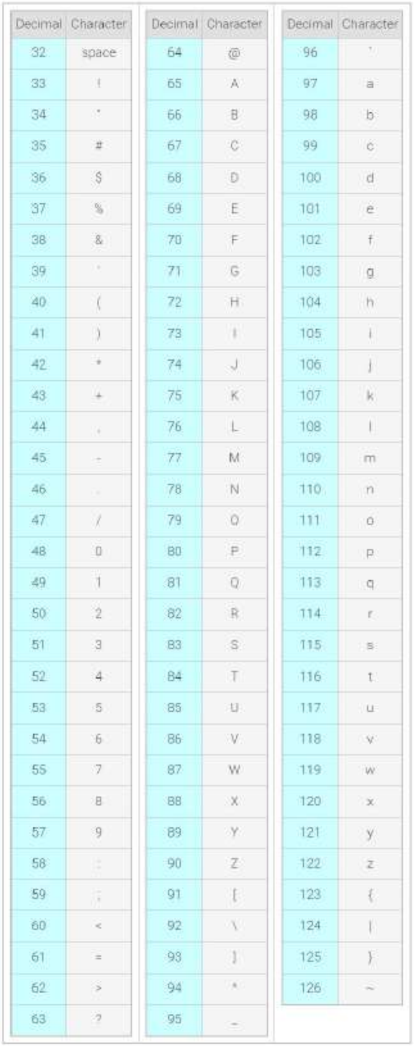
The two-item sequence is called an escape sequence



```
my_string = 'This is a \n \'normal\' string\n'
my_raw_string = r'This is a \n \'raw\' string'

print(my_string)
print(my_raw_string)
```

```
This is a
 'normal' string

This is a \n \'raw\' string
```

Escape sequences can be ignored using a raw string: created by adding an 'r' before a string literal



Decimal	Character	Decimal	Character	Decimal	Character
32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

The built-in function ord() returns an encoded integer value for a string of length one. The built-in function chr() returns a string of one character for an encoded integer

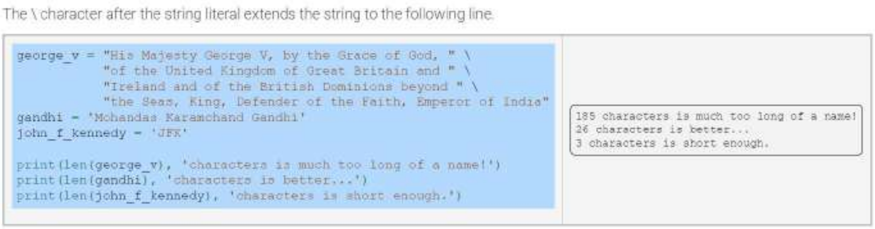
Types

String Basics

A string is a sequence of characters. A string literal is a string value specified in the source code of a program

The string type is a special construct known as a sequence type

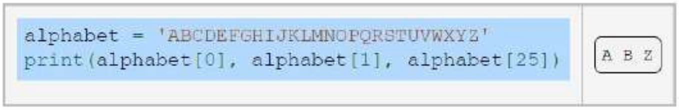
The len() built-in function can be used to find the length of a string (and any other sequence type)



The \ character after the string literal extends the string to the following line.

```
george_v = "His Majesty George V, by the Grace of God, " \
           "of the United Kingdom of Great Britain and " \
           "Ireland and of the British Dominions beyond " \
           "the Seas, King, Defender of the Faith, Emperor of India"
gandhi = 'Mohandas Karamchand Gandhi'
john_f_kennedy = 'JFK'

print(len(george_v), 'characters is much too long of a name!')
print(len(gandhi), 'characters is better...')
print(len(john_f_kennedy), 'characters is short enough.')
```

```
185 characters is much too long of a name!
26 characters is better...
3 characters is short enough.
```



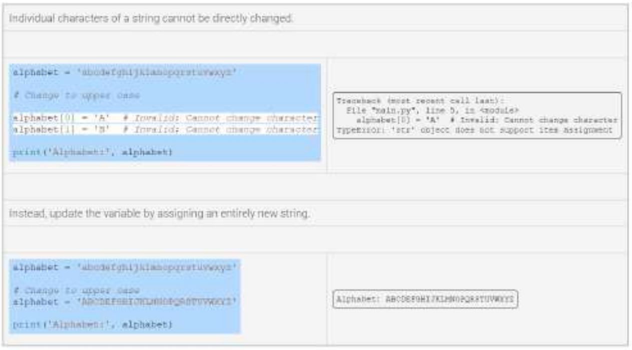
```
alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
print(alphabet[0], alphabet[1], alphabet[25])
```

```
A B Z
```

A sequence starts at 0 from the leftmost character. A character at a specific index can be accessed by appending brackets [ ]

Negative indices can be used to access characters starting from the rightmost character of the string, instead of the leftmost

alphabet[-1] is z



Individual characters of a string cannot be directly changed.

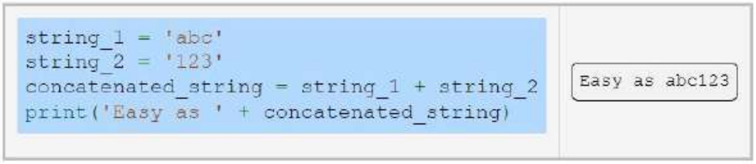
```
alphabet = 'abcdefghijklmnopqrstuvwxyz'

# Change to upper case

alphabet[0] = 'A'  # Invalid: Cannot change character
alphabet[1] = 'B'  # Invalid: Cannot change character

print('Alphabet:', alphabet)
```

```
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    alphabet[0] = 'A'  # Invalid: Cannot change character
TypeError: 'str' object does not support item assignment
```

Instead, update the variable by assigning an entirely new string.

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'

# Change to upper case
alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

print('Alphabet:', alphabet)
```

```
Alphabet: ABCDEFGHIJKLMNOPQRSTUVWXYZ
```



```
string_1 = 'abc'
string_2 = '123'
concatenated_string = string_1 + string_2
print('Easy as ' + concatenated_string)
```

```
Easy as abc123
```

A program can add new characters to the end of a string in a process known as **string concatenation**

The + operator concatenates two strings together

List Basics

A **container** is a construct used to group related values together and contains references to other objects instead of data. A **list** is a container created by surronding a sequence of variables or literals with brackets []

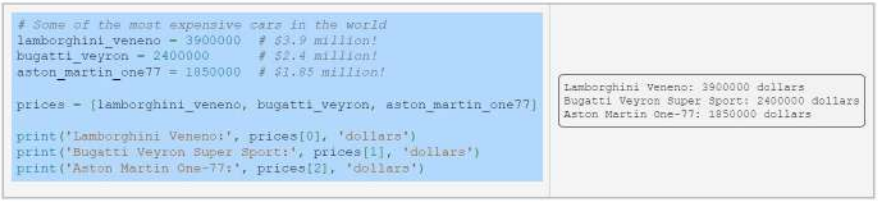
my_list = [10, 'abc'] creates a new list variable my_list that contains the two items: 10 and 'abc'. A list item is called an **element**

A list is also a sequence, meaning the contained elements are ordered by position in the list, known as the element's **index**, starting with 0

my_list = [] creates an empty list

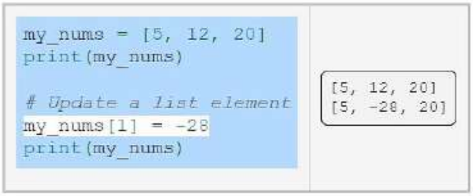
Individual list elements can be accessed by using brackets as in my_list[i], where i is an integer

Figure 4.2.1: Access list elements using an indexing expression.



```
# Some of the most expensive cars in the world
lamborghini_veneno = 3900000  # $3.9 million!
bugatti_veyron = 2400000      # $2.4 million!
aston_martin_one77 = 1850000  # $1.85 million!

prices = [lamborghini_veneno, bugatti_veyron, aston_martin_one77]

print('Lamborghini Veneno:', prices[0], 'dollars')
print('Bugatti Veyron Super Sport:', prices[1], 'dollars')
print('Aston Martin One-77:', prices[2], 'dollars')
```

```
Lamborghini Veneno: 3900000 dollars
Bugatti Veyron Super Sport: 2400000 dollars
Aston Martin One-77: 1850000 dollars
```



```
my_nums = [5, 12, 20]
print(my_nums)

# Update a list element
my_nums[1] = -28
print(my_nums)
```

```
[5, 12, 20]
[5, -28, 20]
```

An element can be updated with a new value by performing an assignment to a position in the list

Adding elements to a list:

- list.append(value): Adds value to the end of list. Ex: `my_list.append('abc')`

Removing elements from a list:

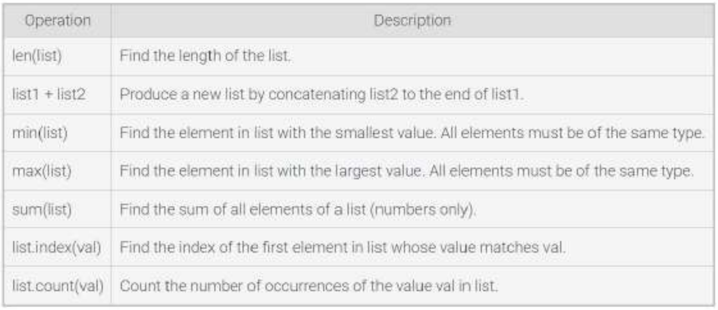
- list.pop(i): Removes the element at index i from list. Ex: `my_list.pop(1)`
- list.remove(v): Removes the first element whose value is v. Ex: `my_list.remove('abc')`

A **method** instructs an object to perform some action, and is executed by specifying the method name following a "." symbol and an object

The **append()** list method is used to add new elements to a list

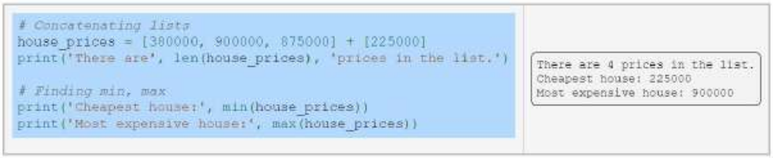
Elements can be removed using the **pop()** or **remove()** methods

Sequence-type functions are built-in functions that operate on sequences like lists and strings.



Operation	Description
len(list)	Find the length of the list.
list1 + list2	Produce a new list by concatenating list2 to the end of list1.
min(list)	Find the element in list with the smallest value. All elements must be of the same type.
max(list)	Find the element in list with the largest value. All elements must be of the same type.
sum(list)	Find the sum of all elements of a list (numbers only).
list.index(val)	Find the index of the first element in list whose value matches val.
list.count(val)	Count the number of occurrences of the value val in list.

Sequence-type methods are methods built into the class definitions of sequences like lists and strings

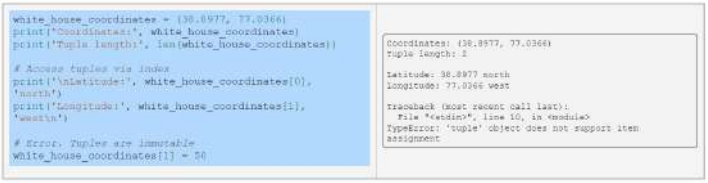


```python
# Concatenating lists
house_prices = [380000, 900000, 875000] + [225000]
print('There are', len(house_prices), 'prices in the list.')

# Finding min, max
print('Cheapest house:', min(house_prices))
print('Most expensive house:', max(house_prices))
```

```
There are 4 prices in the list.
Cheapest house: 225000
Most expensive house: 900000
```

Tuple Basics

A **tuple** behaves similar to a list but is immutable - once created it cannot be changed

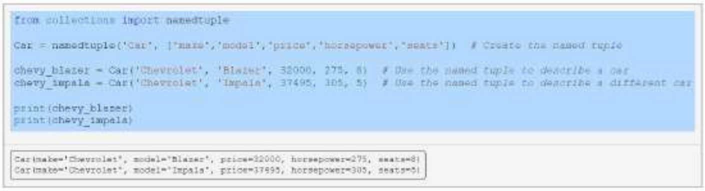


```python
white_house_coordinates = (38.8977, 77.0366)
print('Coordinates:', white_house_coordinates)
print('Tuple length:', len(white_house_coordinates))

# Access tuples via index
print('\nLatitude:', white_house_coordinates[0],
'north')
print('Longitude:', white_house_coordinates[1],
'west\n')

# Error. Tuples are immutable
white_house_coordinates[1] = 50
```

```
Coordinates: (38.8977, 77.0366)
Tuple length: 2

Latitude: 38.8977 north
Longitude: 77.0366 west

Traceback (most recent call last):
  File "<stdin>", line 10, in <module>
TypeError: 'tuple' object does not support item
assignment
```

A new tuple is generated by creating a list of comma-separated values such as 5, 15, 20

A **named tuple** allows the programmer to define a new sample data that consists of named attributes



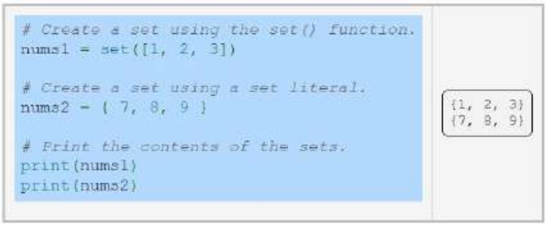
```
from collections import namedtuple

Car = namedtuple('Car', ['make','model','price','horsepower','seats'])  # Create the named tuple

chevy_blazer = Car('Chevrolet', 'Blazer', 32000, 275, 8)  # Use the named tuple to describe a car
chevy_impala = Car('Chevrolet', 'Impala', 37495, 305, 5)  # Use the named tuple to describe a different car

print(chevy_blazer)
print(chevy_impala)
```

```
Car(make='Chevrolet', model='Blazer', price=32000, horsepower=275, seats=8)
Car(make='Chevrolet', model='Impala', price=37495, horsepower=305, seats=5)
```

The **namedtuple** container must be imported to create a new named tuple

Set Basics

A **set** is an unordered collection of unique elements



```
# Create a set using the set() function.
nums1 = set([1, 2, 3])

# Create a set using a set literal.
nums2 = { 7, 8, 9 }

# Print the contents of the sets.
print(nums1)
print(nums2)
```

```
{1, 2, 3}
{7, 8, 9}
```

A set can be created using the **set()** function, which accepts a sequence-type iterable object (list, tuple, string, etc.) whose elements are inserted into the set

A **set literal** can be written using curly brackets { } with commas separating set elements. Note that an empty set can only be created using set()

Creating a set() removes any duplicate values

Because the elements of a set are unordered and have no meaningful position in the collection, the index operator is not valid. Attempting to access the element of a set by position, for example `nums1[2]` to access the element at index 2, is invalid and will produce a runtime error.

A set is often used to reduce a list of items that potentially contains duplicates into a collection of unique values. Simply passing a list into `set()` will cause any duplicates to be omitted in the created set.

Sets are mutable - elements can be added or removed using set methods

Operation	Description
len(set)	Find the length (number of elements) of the set.
set1.update(set2)	Adds the elements in set2 to set1.
set.add(value)	Adds value into the set.
set.remove(value)	Removes value from the set. Raises KeyError if value is not found.
set.pop()	Removes a random element from the set.
set.clear()	Clears all elements from the set.

The add() method places a new element into the set if the set does not contain an element with the provided value

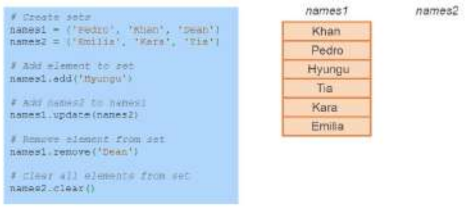
The remove() and pop() methods remove an element from the set

Adding elements to a set:

- set.add(value): Add value into the set. Ex: `my_set.add('abc')`

Remove elements from a set:

- set.remove(value): Remove the element with given value from the set. Raises KeyError if value is not found. Ex: `my_set.remove('abc')`
- set.pop(): Remove a random element from the set. Ex: `my_set.pop()`



```
# Create sets
names1 = {'Pedro', 'Khan', 'Dean'}
names2 = {'Emilia', 'Kara', 'Tia'}

# Add element to set
names1.add('Hyungu')

# Add names2 to names1
names1.update(names2)

# Remove element from set
names1.remove('Dean')

# Clear all elements from set
names2.clear()
```

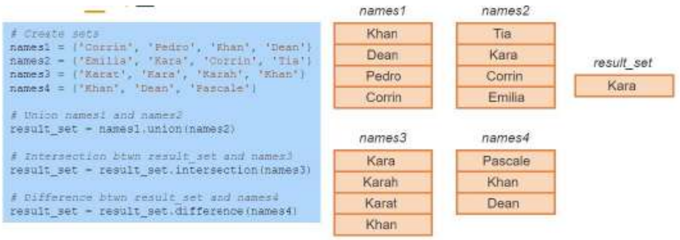
The clear() method removes all elements from a set, leaving the set with a length of 0.

Captions ^

1. Sets can be created using braces {} with commas separating the elements.
2. The add() method adds a single element to a set.
3. The update() method adds the elements of one set to another set.
4. The remove() method removes a single element from a set.
5. The clear() method removes all elements from a set, leaving the set with a length of 0.

Python set objects support typical set theory operations like intersections and unions

Operation	Description
set.intersection(set_a, set_b, set_c...)	Returns a new set containing only the elements in common between `set` and all provided sets.
set.union(set_a, set_b, set_c...)	Returns a new set containing all of the unique elements in all sets.
set.difference(set_a, set_b, set_c...)	Returns a set containing only the elements of `set` that are not found in any of the provided sets.
set_a.symmetric_difference(set_b)	Returns a set containing only elements that appear in exactly one of `set_a` or `set_b`



```
# Create sets
names1 = {'Corrin', 'Pedro', 'Khan', 'Dean'}
names2 = {'Emilia', 'Kara', 'Corrin', 'Tia'}
names3 = {'Karat', 'Kara', 'Karah', 'Khan'}
names4 = {'Khan', 'Dean', 'Pascale'}

# Union names1 and names2
result_set = names1.union(names2)

# Intersection btwn result_set and names3
result_set = result_set.intersection(names3)

# Difference btwn result_set and names4
result_set = result_set.difference(names4)
```

The difference() method builds a set that contains elements only found in result_set that are not in names4.

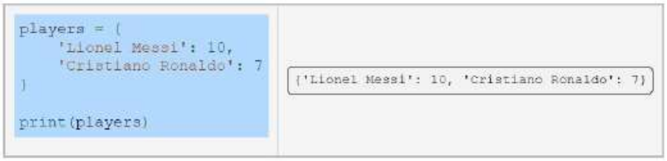
Captions ^

1. The union() method builds a set containing the unique elements from names1 and names2. 'Corrin' only appears once in the resulting set.
2. The intersection() method builds a set that contains all common elements between result_set and names3.
3. The difference() method builds a set that contains elements only found in result_set that are not in names4.

Dictionary Basics

A dictionary is a Python container used to describe associative relationships. A dictionary is represented by the dict object type. A dictionary associates (or "maps") keys with values. A key is a term that can be located in a dictionary, such as the word "cat" in the English dictionary. A value describes some data associated with a key, such as a definition. A key can be any immutable type, such as a number, string, or tuple; a value can be any type

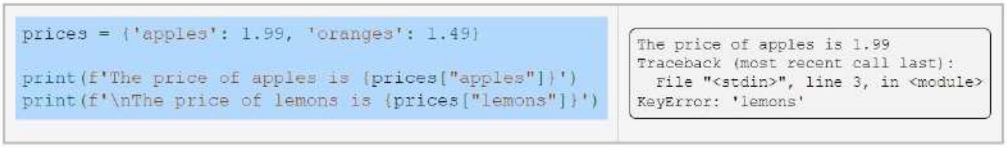
A dict object is created using curly brackets {} to surround the key: value pairs that compromise the dictionary contents



```
players = {
    'Lionel Messi': 10,
    'Cristiano Ronaldo': 7
}

print(players)
```

```
{'Lionel Messi': 10, 'Cristiano Ronaldo': 7}
```

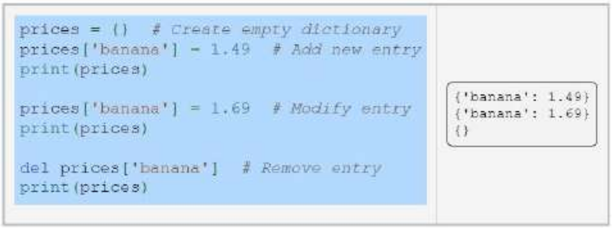
players = {'Lionel Messi': 10, 'Cristiano Ronald': 7 } creates a dictionary called players with two keys: 'Lionel Messi' and 'Cristiano Ronaldo', associated with the values 10 and 7



```
prices = {'apples': 1.99, 'oranges': 1.49}

print(f'The price of apples is {prices["apples"]}')
print(f'\nThe price of lemons is {prices["lemons"]}')
```

```
The price of apples is 1.99
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
KeyError: 'lemons'
```

Though dictionaries maintain a left-to-right ordering, dictionary entries cannot be accessed by indexing. To access an entry, the key is specified in brackets []. If no entry with a matching key exists in the dictionary, then a KeyError runtime error occurs and the program is terminated

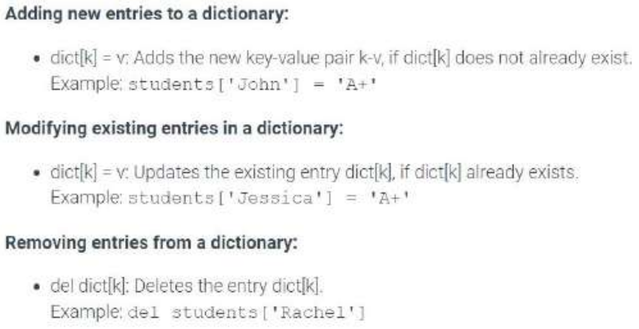


```
prices = {}   # Create empty dictionary
prices['banana'] = 1.49   # Add new entry
print(prices)

prices['banana'] = 1.69   # Modify entry
print(prices)

del prices['banana']   # Remove entry
print(prices)
```

```
{'banana': 1.49}
{'banana': 1.69}
{}
```

A new dictionary entry is added by using brackets to specify the key:
prices['banana'] = 1.49

The del keyword is used to remove entries from a dictionary:
del prices['papaya']
removes the entry whose key is 'papaya'



Adding new entries to a dictionary:

- dict[k] = v: Adds the new key-value pair k-v, if dict[k] does not already exist.
  Example: `students['John'] = 'A+'`

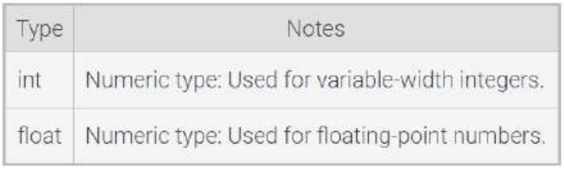
Modifying existing entries in a dictionary:

- dict[k] = v: Updates the existing entry dict[k], if dict[k] already exists.
  Example: `students['Jessica'] = 'A+'`

Removing entries from a dictionary:

- del dict[k]: Deletes the entry dict[k].
  Example: `del students['Rachel']`

Common Data Types Summary



Type	Notes
int	Numeric type: Used for variable-width integers.
float	Numeric type: Used for floating-point numbers.

Numeric types int and float represent the most common types used to store data

Sequence types string, list, and tuple are all containers for collections of objects ordered by position in the sequence, where the first object has an index of 0 and subsequent elements have indices 1, 2, etc.

The only mapping type in Python is the dict type. Each element of a dict is independent, having no special ordering or relation to other elements

Type	Notes
string	Sequence type: Used for text.
list	Sequence type: A mutable container with ordered elements.
tuple	Sequence type: An immutable container with ordered elements.
set	Set type: A mutable container with unordered and unique elements.
dict	Mapping type: A container with key-values associated elements.

Choosing a container type

New programmers often struggle with choosing the types that best fit their needs, such as choosing whether to store particular data using a list, tuple, or dict. In general, a programmer might use a list when data has an order, such as lines of text on a page. A programmer might use a tuple instead of a list if the contained data should not change. If order is not important, a programmer might use a dictionary to capture relationships between elements, such as student names and grades.

Type Conversions

A type conversion is an conversion of one data type to another, such as an integer to a float

Implicit Conversion:

- For an arithmetic operator like + or *, if either is a float, the other is automatically converted to float, and then a floating-point operation is performed
- For assignments, the right side type is converted to the left side type

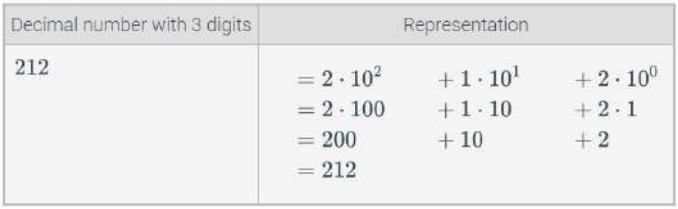
integer-to-float conversion is straightforward: 25 becomes 25.0

float-to-integer conversion just drops the fraction: 4.9 becomes 4

Function	Notes	Can convert:
int()	Creates integers	int, float, strings w/ integers only
float()	Creates floats	int, float, strings w/ integers or fractions
str()	Creates strings	Any

Binary Numbers

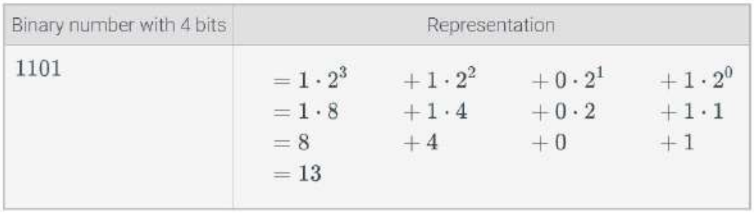
Because each memory location is composed of bits (0s and 1s), a processor stores a number using base 2, known as **binary number**



Decimal number with 3 digits	Representation		
212	$= 2 \cdot 10^2$	$+ 1 \cdot 10^1$	$+ 2 \cdot 10^0$
	$= 2 \cdot 100$	$+ 1 \cdot 10$	$+ 2 \cdot 1$
	$= 200$	$+ 10$	$+ 2$
	$= 212$		

For a number in the more familiar base 10, known as **decimal number**, each digit must be 0-9 and each digit's place is weighed by increasing powers of 10

In **base 2**, each digit must be 0-1 and each digit's place is weighed by increasing powers of 2

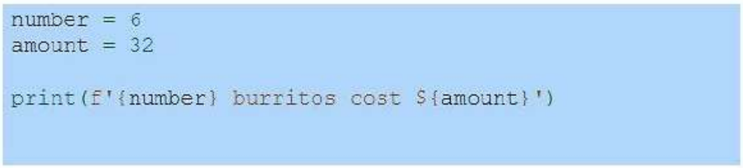


Binary number with 4 bits	Representation			
1101	$= 1 \cdot 2^3$	$+ 1 \cdot 2^2$	$+ 0 \cdot 2^1$	$+ 1 \cdot 2^0$
	$= 1 \cdot 8$	$+ 1 \cdot 4$	$+ 0 \cdot 2$	$+ 1 \cdot 1$
	$= 8$	$+ 4$	$+ 0$	$+ 1$
	$= 13$			

String Formatting

A **formatted string literal** or **f-string** allows a programmer to create a string with placeholder expressions that are evaluated as the program executes

A placeholder expression is also called a **replacement field**, as its value replaces the expression in the final output



```
number = 6
amount = 32

print(f'{number} burritos cost ${amount}')
```

6 burritos cost $32

Additional f-string features

An = sign can be provided after the expression in a replacement field to print both the expression and its result which is a useful debugging technique when dynamically generating lots of strings and output. Ex: `f'{2*4=}'` produces the string "2*4=8".

Additionally, double braces `{{` and `}}` can be used to place an actual curly brace into an f-string. Ex: `f'{{Jeff Bezos}}: Amazon'` produces the string "{Jeff Bezos}: Amazon".

Table 4.10.1: f-string examples.

Example	Output
`print(f'{2**2=}')`	`2**2=4`

`two_power_two = 2**2`
`print(f'{two_power_two=}')`
	`two_power_two=4`

`print(f'{2**2=},{2**4=}')`	`2**2=4,2**4=16`
`print(f'{{2**2}}')`	`{2**2}`
`print(f'{{{2**2=}}}')`	`{2**2=4}`

A **format specification** inside a replacement field allows a value's formatting in the string to be customized

A **presentation type** is a part of a format specification that determines how to represent a value in text form, such as integer (4), floating point (4.0), fixed precision decimal (4.000), percentage (4%), binary (100), etc

Type	Description	Example	Output

s	String (default presentation type - can be omitted)	`name = 'Aiden'`
		`print(f'{name:s}')`
			`Aiden`

d	Decimal (integer values only)	`number = 4`
		`print(f'{number:d}')`
			`4`

b	Binary (integer values only)	`number = 4`
		`print(f'{number:b}')`
			`100`

x, X	Hexadecimal in lowercase (x) and uppercase (X) (integer values only)	`number = 31`
		`print(f'{number:x}')`
			`1f`

e	Exponent notation	`number = 44`
		`print(f'{number:e}')`
			`4.400000e+01`

f	Fixed-point notation (6 places of precision)	`number = 4`
		`print(f'{number:f}')`
			`4.000000`

.[precision]f	Fixed-point notation (programmer-defined precision)	`number = 4`
		`print(f'{number:.2f}')`
			`4.00`

0[precision]d	Leading 0 notation	`number = 4`
		`print(f'{number:03d}')`
			`004`

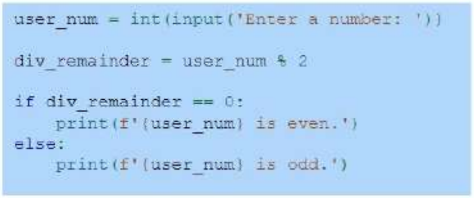
Branching

If-else branches

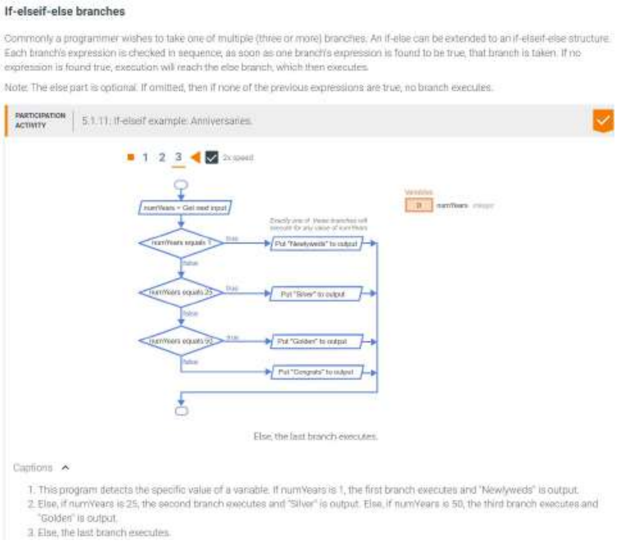
In a program, a **branch** is a sequence of statements only executed under a certain condition

An **if** branch is a branch taken only if an expression is true

An **if-else** branch has two branches: The first branch is executed if an expression is true, else the other branch is executed



```
user_num = int(input('Enter a number: '))

div_remainder = user_num % 2

if div_remainder == 0:
    print(f'{user_num} is even.')
else:
    print(f'{user_num} is odd.')
```



If-elseif-else branches

Commonly a programmer wishes to take one of multiple (three or more) branches. An if-else can be extended to an if-elseif-else structure. Each branch's expression is checked in sequence; as soon as one branch's expression is found to be true, that branch is taken. If no expression is found true, execution will reach the else branch, which then executes.

Note: The else part is optional. If omitted, then if none of the previous expressions are true, no branch executes.

PARTICIPATION ACTIVITY 5.1.11: If-elseif example: Anniversaries.

Else, the last branch executes.

Captions

1. This program detects the specific value of a variable. If numYears is 1, the first branch executes and "Newlyweds" is output.
2. Else, if numYears is 25, the second branch executes and "Silver" is output. Else, if numYears is 50, the third branch executes and "Golden" is output.
3. Else, the last branch executes.

Detecting equal values with branches

An **if** statement executes a group of statements if an expression is true

Equality operators	Description	Example (assume x is 3)
==	a **==** b means a is equal to b	x == 3 is true x == 4 is false
!=	a **!=** b means a is not equal to b	x != 3 is false x != 4 is true

The equality operator (==) evaluates to true if the left and right sides are equal

The inequality operator (!=) evaluates to true if the left and right sides are not equal, or different

An expression involving an equality operator evaluates to a Boolean value. A Boolean is a type that has just two values: true or false

```
hotel_rate = 150

num_years = int(input('Enter years married: '))

if num_years == 50:
    print('Congratulations on 50 years of marriage!')
    hotel_rate = hotel_rate / 2

print(f'Your hotel rate: ${hotel_rate:.2f}')
```

Figure 5.2.1: Multi-branch if-else statement. Only 1 branch will execute.

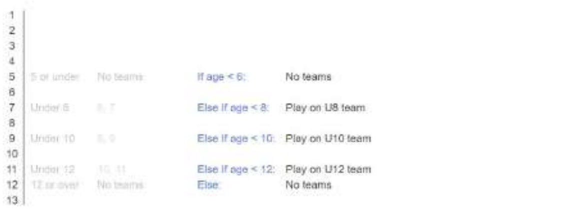
```
if expression1:
   # Statements that execute when expression1 is true
   # (first branch)
elif expression2:
   # Statements that execute when expression1 is false and expression2 is true
   # (second branch)
else:
   # Statements that execute when expression1 is false and expression2 is false
   # (third branch)
```

Figure 5.2.2: Multi-branch if-else example: Anniversaries.

```
num_years = int(input('Enter number years married: '))

if num_years == 1:
    print('Your first year -- great!')
elif num_years == 10:
    print('A whole decade -- impressive.')
elif num_years == 25:
    print('Your silver anniversary -- enjoy.')
elif num_years == 50:
    print('Your golden anniversary -- amazing.')
else:
    print('Nothing special.')
```

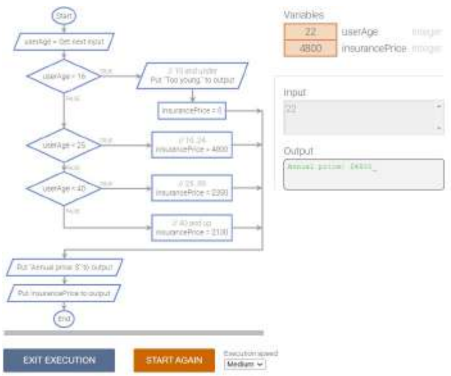
```
Enter number years married: 10
A whole decade -- impressive.

...

Enter number years married: 25
Your silver anniversary -- enjoy.

...

Enter number years married: 30
Nothing special.

...

Enter number years married: 1
Your first year -- great!
```

Detecting ranges with branches



```
5 or under   No teams    If age < 6:        No teams

Under 8      6, 7        Else If age < 8:   Play on U8 team

Under 10     8, 9        Else If age < 10:  Play on U10 team

Under 12     10, 11      Else If age < 12:  Play on U12 team
12 or over   No teams    Else:              No teams
```

Using a sequence of decisions enables a concise way of specifying a range, such as saying U12 rather than saying ages 10 and 11.



Variables

22	userAge	integer
4800	insurancePrice	integer

Input

```
22
```

Output

```
Annual price: $4800_
```

Execution speed: Medium

Captions ^

1. Kids of various ages may wish to play soccer. A soccer club may not have teams for kids 5 and under.
2. One level of teams is listed as "Under 8" (orU8), which is understood to mean just for ages 7 and 6, but not 5 and younger.
3. Likewise, U10 means ages 9 and 8, and U12 means 11 and 10. No teams exist for ages 12 and over.
4. Using a sequence of decisions enables a concise way of specifying a range, such as saying U12 rather than saying ages 10 and 11.

A **relational operator** checks how one operand's value relates to another, like being greater than

Relational operators	Description	Example (assume x is 3)
<	a **<** b means a is less-than b	x < 4 is true x < 3 is false
>	a **>** b means a is greater-than b	x > 2 is true x > 3 is false
<=	a **<=** b means a is less-than-or-equal to b	x <= 4 is true x <= 3 is true x <= 2 is false
>=	a **>=** b means a is greater-than-or-equal to b	x >= 2 is true x >= 3 is true x >= 4 is false

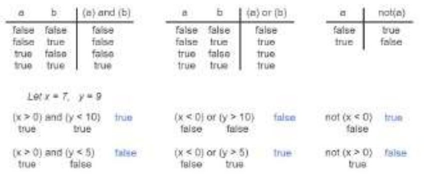
Python supports **operator chaining**

$a < b < c$

Detecting ranges using logical operators

A **logical operator** treats operands as being true or false, and evaluates to true or false. Logical operators include: and, or, not

Logical operator	Description
(a) and (b)	***Logical and***: true when both operands are true
(a) or (b)	***Logical or***: true when at least one of the two operands is true
not(a)	***Logical not***: true when the one operand is false, and vice-versa



a	b	(a) and (b)
false	false	false
false	true	false
true	false	false
true	true	true

a	b	(a) or (b)
false	false	false
false	true	true
true	false	true
true	true	true

a	not(a)
false	true
true	false

Let x = 7, y = 9

(x > 0) and (y < 10) — true
true — true

(x > 0) and (y < 5) — false
true — false

(x < 0) or (y > 10) — false
false — false

(x < 0) or (y > 5) — true
false — true

not (x < 0) — true
false

not (x > 0) — false
true

Each operand is an expression. If x = 7, y = 9, then (x > 0) and (y < 10) is true and true, so the and operator evaluates to true (both operands are true).

Captions ^

1. The and operator evaluates to true only if BOTH operands are true. Each operand (a, b above) is an expression that evaluates to true or false.
2. The or operator evaluates to true if ANY operand is true (a, b, or both).
3. The not operator evaluates to the opposite of the operand.
4. Each operand is an expression. If x = 7, y = 9, then (x > 0) and (y < 10) is true and true, so the and operator evaluates to true (both operands are true).

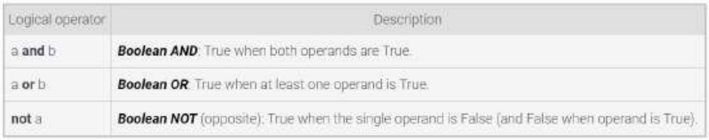
If no gaps exist in the series of ranges, the range is **Implicit**

If gabs are in the series of ranges, the range is **Explicit**

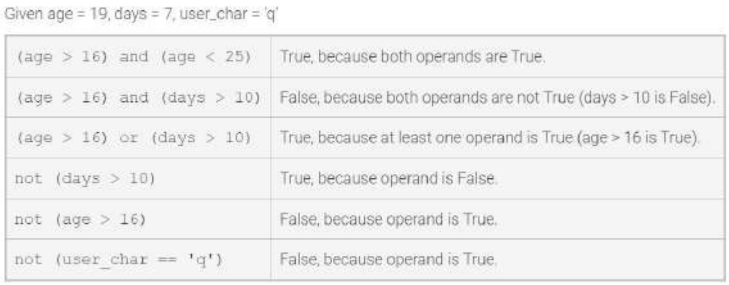
A **Boolean** refers to a value that is either True or False

Figure 5.5.1: Creating a Boolean.

```
my_bool = True     # Assigns my_bool with the boolean value True

is_small = my_val < 3   # Assigns is_small with the result of the expression (False)
```



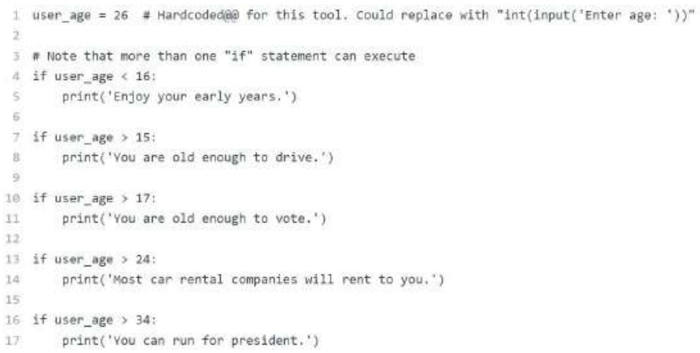
Logical operator	Description
a **and** b	***Boolean AND***: True when both operands are True.
a **or** b	***Boolean OR***: True when at least one operand is True.
not a	***Boolean NOT*** (opposite): True when the single operand is False (and False when operand is True).



Given age = 19, days = 7, user_char = 'q'

`(age > 16) and (age < 25)`	True, because both operands are True.
`(age > 16) and (days > 10)`	False, because both operands are not True (days > 10 is False).
`(age > 16) or (days > 10)`	True, because at least one operand is True (age > 16 is True).
`not (days > 10)`	True, because operand is False.
`not (age > 16)`	False, because operand is True.
`not (user_char == 'q')`	False, because operand is True.

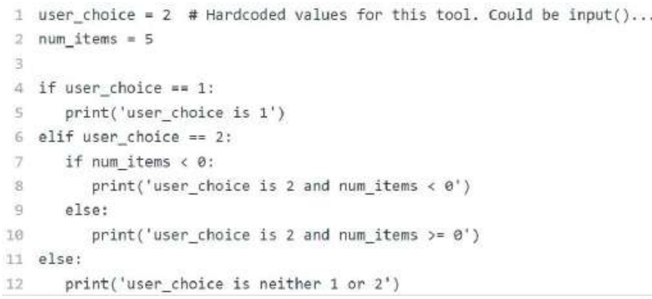
Detecting multiple features with branches



```
user_age = 26  # Hardcoded@@ for this tool. Could replace with "int(input('Enter age: '))"

# Note that more than one "if" statement can execute
if user_age < 16:
    print('Enjoy your early years.')

if user_age > 15:
    print('You are old enough to drive.')

if user_age > 17:
    print('You are old enough to vote.')

if user_age > 24:
    print('Most car rental companies will rent to you.')

if user_age > 34:
    print('You can run for president.')
```

Multiple if statements are considered independently from each other, so multiple branches may execute

A branch's statements can include any valid statements, including another if-else statement known as **nested if-else statements**



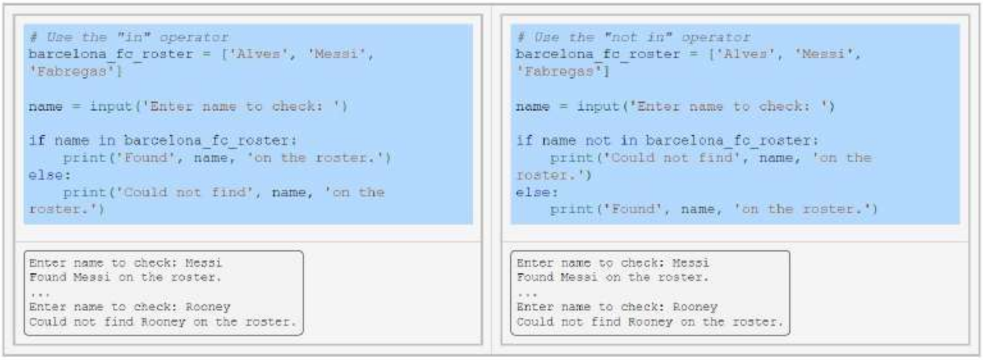
```
user_choice = 2  # Hardcoded values for this tool. Could be input()...
num_items = 5

if user_choice == 1:
   print('user_choice is 1')
elif user_choice == 2:
   if num_items < 0:
      print('user_choice is 2 and num_items < 0')
   else:
      print('user_choice is 2 and num_items >= 0')
else:
   print('user_choice is neither 1 or 2')
```

Comparison of values with the same type, such as `5 < 2`, or `'abc' >= 'ABCDEF'`, depends on the types being compared.

- Numbers are arithmetically compared.
- Strings are compared by converting each character to a number value (ASCII or Unicode), and then comparing each character in order. Most string comparisons use equality operators "==" or "!=", as in `today == 'Friday'`.
- Lists and tuples are compared via an ordered comparison of every element in the sequence. Every element between the sequences must compare as equal for an equality operator to evaluate to True. Relational operators like < or > can also be used: The result is determined by the first mismatching elements in the sequences. For example, if x = [1, 5, 2] and y = [1, 4, 3], then evaluating `x < y` first evaluates that 1 and 1 match. Since the first list elements match, neither list can be considered to be less than the other, nor can the lists be declared equal without comparing more elements. So the next elements must be compared. The next elements do not match, so `5 < 4` is evaluated, which produces a value of False.
- Dictionaries are compared only with == and !=. To be equal, two dictionaries must have the same set of keys and the same corresponding value for each key.

Membership and identity operators

The **in** and **not in** operators, known as **membership operators**, yield True or False if the left operhand matches the value of some element in the right operhand, which is always a container



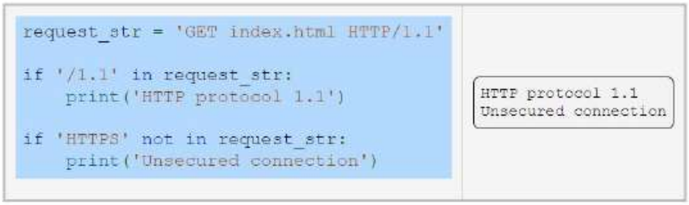
```
# Use the "in" operator
barcelona_fc_roster = ['Alves', 'Messi',
'Fabregas']

name = input('Enter name to check: ')

if name in barcelona_fc_roster:
    print('Found', name, 'on the roster.')
else:
    print('Could not find', name, 'on the
roster.')
```

```
Enter name to check: Messi
Found Messi on the roster.
...
Enter name to check: Rooney
Could not find Rooney on the roster.
```

```
# Use the "not in" operator
barcelona_fc_roster = ['Alves', 'Messi',
'Fabregas']

name = input('Enter name to check: ')

if name not in barcelona_fc_roster:
    print('Could not find', name, 'on the
roster.')
else:
    print('Found', name, 'on the roster.')
```

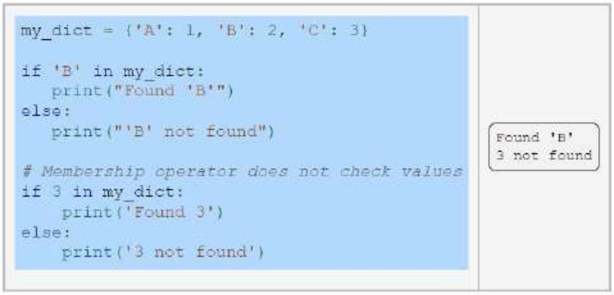
```
Enter name to check: Messi
Found Messi on the roster.
...
Enter name to check: Rooney
Could not find Rooney on the roster.
```

Membership operators can be used to check whether a string is a **substring**, or matching subset of characters, of a larger string



```
request_str = 'GET index.html HTTP/1.1'

if '/1.1' in request_str:
    print('HTTP protocol 1.1')

if 'HTTPS' not in request_str:
    print('Unsecured connection')
```

```
HTTP protocol 1.1
Unsecured connection
```



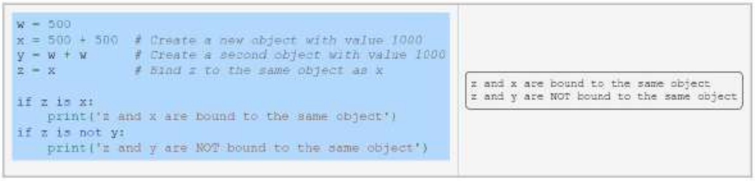
```
my_dict = {'A': 1, 'B': 2, 'C': 3}

if 'B' in my_dict:
   print("Found 'B'")
else:
   print("'B' not found")

# Membership operator does not check values
if 3 in my_dict:
    print('Found 3')
else:
    print('3 not found')
```

```
Found 'B'
3 not found
```

Checking for membership in a dict

Membership in a dictionary implies that a specific key exists in the dictionary

To determine whether two variables are the same object, the identity operator, is, can be used to check whether two operands are bound to a single object. The inverse identity operator, is not, gives the negated value of 'is'

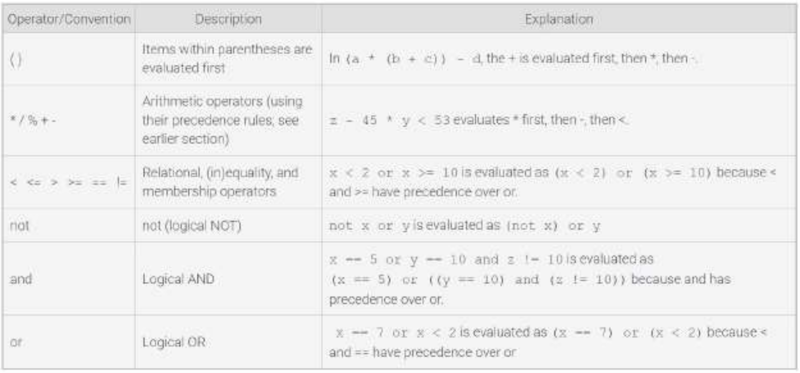


```
w = 500
x = 500 + 500  # Create a new object with value 1000
y = w + w      # Create a second object with value 1000
z = x          # Bind z to the same object as x

if z is x:
    print('z and x are bound to the same object')
if z is not y:
    print('z and y are NOT bound to the same object')
```

```
z and x are bound to the same object
z and y are NOT bound to the same object
```

if x is y is True, then x is not y is False

An expression is evaluated in programming using these precedence rules

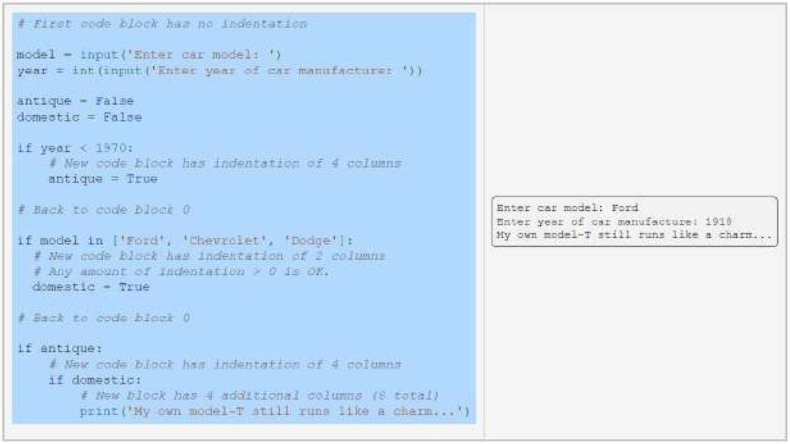


Operator/Convention	Description	Explanation
( )	Items within parentheses are evaluated first	In (a * (b + c)) - d, the + is evaluated first, then *, then -.
* / % + -	Arithmetic operators (using their precedence rules; see earlier section)	z - 45 * y < 53 evaluates * first, then -, then <.
< <= > >= == !=	Relational, (in)equality, and membership operators	x < 2 or x >= 10 is evaluated as (x < 2) or (x >= 10) because < and >= have precedence over or.
not	not (logical NOT)	not x or y is evaluated as (not x) or y
and	Logical AND	x == 5 or y == 10 and z != 10 is evaluated as (x == 5) or ((y == 10) and (z != 10)) because and has precedence over or.
or	Logical OR	x == 7 or x < 2 is evaluated as (x == 7) or (x < 2) because < and == have precedence over or

Good practice is to use parentheses in expressions to make the intended order of evaluation explicit

Code blocks and indentation

A code block is a series of statements that are grouped together defined by its indentation level



```
# First code block has no indentation

model = input('Enter car model: ')
year = int(input('Enter year of car manufacture: '))

antique = False
domestic = False

if year < 1970:
    # New code block has indentation of 4 columns
    antique = True

# Back to code block 0

if model in ['Ford', 'Chevrolet', 'Dodge']:
  # New code block has indentation of 2 columns
  # Any amount of indentation > 0 is OK.
  domestic = True

# Back to code block 0

if antique:
    # New code block has indentation of 4 columns
    if domestic:
        # New block has 4 additional columns (8 total)
        print('My own model-T still runs like a charm...')
```

```
Enter car model: Ford
Enter year of car manufacture: 1918
My own model-T still runs like a charm...
```

The number of columns of text considered to be acceptable varies from 80 to 120

Multiple lines enclosed within parentheses are implicitly joined into a single string (without newlines between each line); use implicit line joining for very long strings or functions with numerous arguments. Ex: All extra lines are indented to the same column as the opening quotation mark on the first line.

When declaring list or dict literals, entries can be placed on separate lines for clarity.

```
declaration = ("When in the Course of human events, it becomes necessary for "
               "one people to dissolve the political bands which have connected "
               "them with another, and to assume among the powers of the earth...")
```

```
result_of_power = math.pow(long_variable_name_left_operand,
                           long_variable_name_right_operand)
```

Containers like lists and dicts can be broken into multiple lines, with the elements on separate, indented lines.

```
my_list = [
    1, 2, 3,
    4, 5, 6
    ]
```

```
my_dict = {
    'entryA': 1,
    'entryB': 2
}
```

Conditional expressions

A conditional expression has the following form:

```
expr_when_true if condition else expr_when_false
```

A conditional expression has three operands and thus is sometimes referred to as a ternary operation

All three operands are expressions. The condition in the middle is evaluated first. If the condition evaluates to True, then expr_when_true is evaluated. If the condition evaluates to False, then expr_when_false is evaluated. For example, if x is 2, then the conditional expression `5 if x==2 else 9*x` evaluates to 5.

Good practice is to restrict usage of conditional expressions to an assignment statement, as in: `y = 5 if (x == 2) else 9*x`. Some Python programmers denounce conditional expressions as difficult to read and comprehend, since the middle operand is actually the first evaluated, and left-to-right syntax is preferred. However, simple assignments such as the statement above are acceptable.

```
if condition:
    my_var = expr1
else:
    my_var = expr2
```

```
my_var = expr1 if (condition) else  expr2
```

If the condition evaluates to False, then expr2 is evaluated and my_var is assigned with expr2's value.

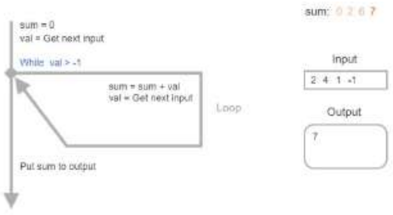
Captions ^

1. This if-else form can be written as a conditional expression.
2. The condition in the middle is evaluated first.
3. If the condition evaluates to True, then expr1 is evaluated and my_var is assigned with expr1's value.
4. If the condition evaluates to False, then expr2 is evaluated and my_var is assigned with expr2's value.

Loops

A **loop** is a program construct that repeatedly executes the loop's statements (known as the **loop body**) while the loop's expression is true; when false, execution proceeds past the loop

Each time through a loop's statements is called an **iteration**



The next input is -1. This time, -1 > -1 is false, so the loop is not entered. Instead, execution proceeds past the loop, where a statement puts sum, which is 7, to the output.

Captions ^

1. A loop is like a branch, but jumping back to the expression when done. Thus, the loop's statements may execute multiple times, before execution proceeds past the loop.
2. This program gets an input value. If the value > -1, the program adds the value to a sum, gets another input, and repeats. val is 2, so the loop's statements execute, making sum 2.
3. The loop's statements ended by getting the next input, which is 4. The loop's expression 4 > -1 is true, so the loop's statements execute again, making sum 2 + 4 or 6.
4. The loop's statements got the next input of 1. The loop's expression 1 > -1 is true, so the loop's statements execute a third time, making sum 6 + 1 or 7.
5. The next input is -1. This time, -1 > -1 is false, so the loop is not entered. Instead, execution proceeds past the loop, where a statement puts sum, which is 7, to the output.

While loop

```
while expression:  # Loop expression
    # Loop body: Sub-statements to execute
    # if the loop expression evaluates to True

# Statements to execute after the expression evaluates to False
```

A while loop is a construct that repeatedly executes an idented block of code as long as the loop's expression is True

```
curr_power = 2
user_char = 'y'

while user_char == 'y':
    print(curr_power)
    curr_power = curr_power * 2
    user_char = input()

print('Done')
```

The following example uses the statement while user_value != 'q': to allow a user to end a face-drawing program by entering the character 'q'. The letter in this case is a sentinel value, a value that when evaluated by the loop expression causes the loop to terminate

```
nose = '0'  # Looks a little like a nose
user_value = '-'

while user_value != 'q':
    print(f' {user_value} {user_value} ')  # Print eyes
    print(f'  {nose}  ')  # Print nose
    print(user_value*5)  # Print mouth
    print('\n')

    # Get new character for eyes and mouth
    user_input = input("Enter a character ('q' for quit): \n")
    user_value = user_input[0]

print('Goodbye.\n')
```

```
 - -
  0
-----


Enter a character ('q' for quit): x
 x x
  0
xxxxx


Enter a character ('q' for quit): @
 @ @
  0
@@@@@


Enter a character ('q' for quit): q
Goodbye.
```

An infinite loop is a loop that will always execute because the loop's expression is always True

```
'''
Program that has a conversation with the user.
Uses elif branching and a random number to mix up the program's responses.
'''
import random  # Import a library to generate random numbers

print('Tell me something about yourself.')
print('You can type \'Goodbye\' at anytime to quit.\n')
```

The first few lines represent a docstring: a multi-line string literal delimited at the beginning and end by triple quotes

If the two strings are not equal, the while loop body executes. Within the loop body, the program obtains a random number between 0 and 2 by using the random library. The randint() function provides a new random number each time the function is called

Random number between 0 and 5, inclusive:
random.randint(0, 5)

```python
import random
random.seed(5)

keep_bidding = 'y'
next_bid = 0

while keep_bidding != 'n':
    next_bid = next_bid + random.randint(1, 10)
    print(f'I\'ll bid ${next_bid}!')
    print('Continue bidding? (y/n)', end=' ')
    keep_bidding = input()
```

Counting

```python
# Iterating N times using a loop variable
i = 1
while i <= N:
    # Loop body statements go here
    i = i + 1
```

A loop commonly must iterate a specific number of times, such as 10 times

The programmer can use a variable to count the number of iterations, called a loop variable

Figure 6.4.1: While loop with loop variable that counts down.

```python
i = 5
while i >= 1:
    # Loop body statements go here
    i = i - 1
```

Figure 6.4.2: Loop variable increased by 2 per iteration.

```python
i = 0
while i <= 100:
    # Loop body statements go here
    i = i + 2
```

Construct 6.4.2: Operators like += are common in loops.

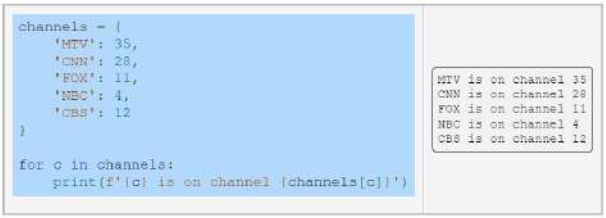
```python
i = 0
while i < N:
    # Loop body statements go here
    i += 1
```

For loops

A **for loop** statement loops over each element in a container one at a time, assigning a variable with the next element that can be used in the loop body

```
for variable in container:
    # Loop body: Sub-statements to execute
    # for each item in the container

# Statements to execute after the for loop is complete
```

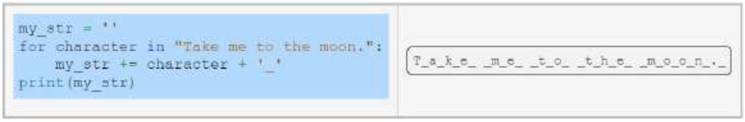
Figure 6.5.1: A for loop assigns the loop variable with a dictionary's keys.



```
channels = {
    'MTV': 35,
    'CNN': 28,
    'FOX': 11,
    'NBC': 4,
    'CBS': 12
}

for c in channels:
    print(f'{c} is on channel {channels[c]}')
```

```
MTV is on channel 35
CNN is on channel 28
FOX is on channel 11
NBC is on channel 4
CBS is on channel 12
```

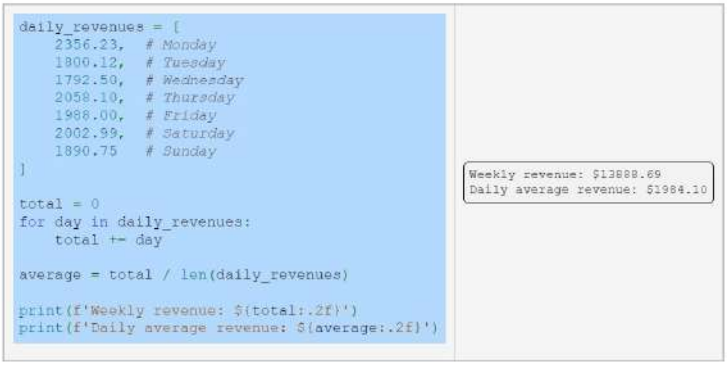
Figure 6.5.2: Using a for loop to access each character of a string.



```
my_str = ''
for character in "Take me to the moon.":
    my_str += character + '_'
print(my_str)
```

```
T_a_k_e_ _m_e_ _t_o_ _t_h_e_ _m_o_o_n_._
```

Figure 6.5.3: For loop example: Calculating shop revenue.

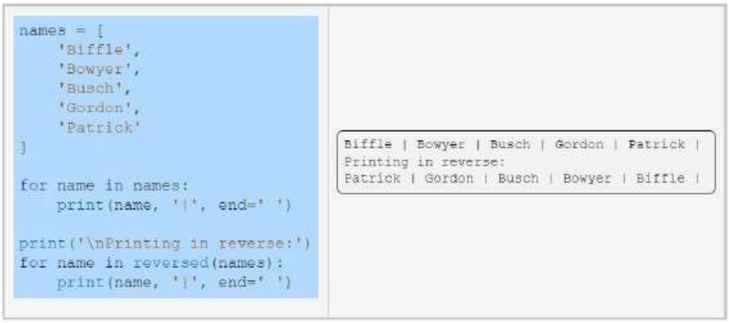


```
daily_revenues = [
    2356.23,  # Monday
    1800.12,  # Tuesday
    1792.50,  # Wednesday
    2058.10,  # Thursday
    1988.00,  # Friday
    2002.99,  # Saturday
    1890.75   # Sunday
]

total = 0
for day in daily_revenues:
    total += day

average = total / len(daily_revenues)

print(f'Weekly revenue: ${total:.2f}')
print(f'Daily average revenue: ${average:.2f}')
```

```
Weekly revenue: $13888.69
Daily average revenue: $1984.10
```

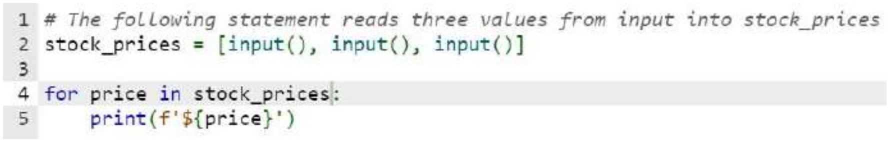
A for loop may also iterate backwards over a sequence, starting at the last element, by using the **reversed()** function to reverse the order of the elements

The following program first prints a list that is ordered alphabetically, then prints the same list in reverse order.

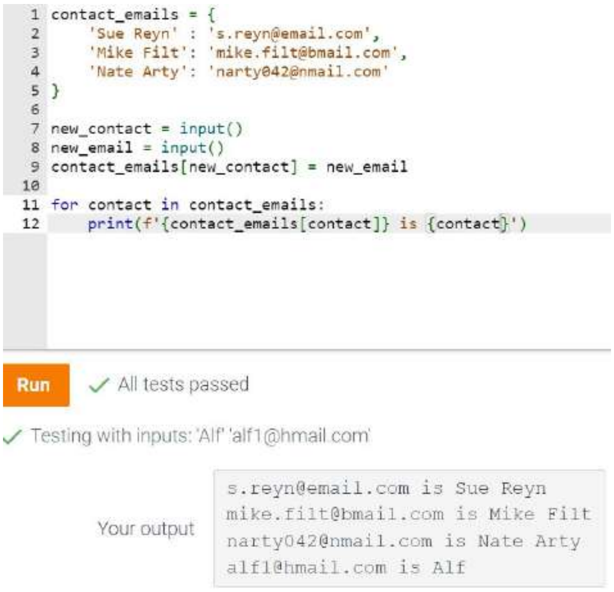


```
names = [
    'Biffle',
    'Bowyer',
    'Busch',
    'Gordon',
    'Patrick'
]

for name in names:
    print(name, '|', end=' ')

print('\nPrinting in reverse:')
for name in reversed(names):
    print(name, '|', end=' ')
```

```
Biffle | Bowyer | Busch | Gordon | Patrick |
Printing in reverse:
Patrick | Gordon | Busch | Bowyer | Biffle |
```



```
# The following statement reads three values from input into stock_prices
stock_prices = [input(), input(), input()]

for price in stock_prices:
    print(f'${price}')
```



```
contact_emails = {
    'Sue Reyn' : 's.reyn@email.com',
    'Mike Filt': 'mike.filt@bmail.com',
    'Nate Arty': 'narty042@nmail.com'
}

new_contact = input()
new_email = input()
contact_emails[new_contact] = new_email

for contact in contact_emails:
    print(f'{contact_emails[contact]} is {contact}')
```

✓ All tests passed

✓ Testing with inputs: 'Alf' 'alf1@hmail.com'

Your output

```
s.reyn@email.com is Sue Reyn
mike.filt@bmail.com is Mike Filt
narty042@nmail.com is Nate Arty
alf1@hmail.com is Alf
```

Counting using the range() function

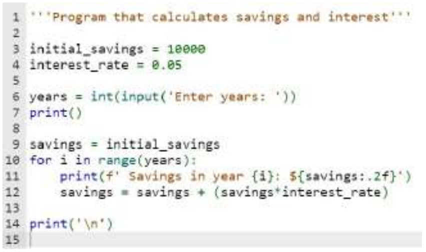
The range() function allows counting in for loops as well. range() generates a sequence of integers between a starting integer that is included in the range, an ending integer that is not included in the range, and an integer step value

The range() function can take up to three integer arguments.

- `range(Y)` generates a sequence of all non-negative integers less than Y.
  Ex: `range(3)` creates the sequence 0, 1, 2.
- `range(X, Y)` generates a sequence of all integers >= X and < Y.
  Ex: `range(-7, -3)` creates the sequence -7, -6, -5, -4.
- `range(X, Y, Z)`, where Z is positive, generates a sequence of all integers >= X and < Y, incrementing by
  Ex: `range(0, 50, 10)` creates the sequence 0, 10, 20, 30, 40.
- `range(X, Y, Z)`, where Z is negative, generates a sequence of all integers <= X and > Y, incrementing by
  Ex: `range(3, -1, -1)` creates the sequence 3, 2, 1, 0.

Table 6.6.1: Using the range() function.

Range	Generated sequence	Explanation
`range(5)`	0 1 2 3 4	Every integer from 0 to 4.
`range(0, 5)`	0 1 2 3 4	Every integer from 0 to 4.
`range(3, 7)`	3 4 5 6	Every integer from 3 to 6.
`range(10, 13)`	10 11 12	Every integer from 10 to 12.
`range(0, 5, 1)`	0 1 2 3 4	Every 1 integer from 0 to 4.
`range(0, 5, 2)`	0 2 4	Every 2nd integer from 0 to 4.
`range(5, 0, -1)`	5 4 3 2 1	Every 1 integer from 5 down to 1
`range(5, 0, -2)`	5 3 1	Every 2nd integer from 5 down to 1



```python
'''Program that calculates savings and interest'''

initial_savings = 10000
interest_rate = 0.05

years = int(input('Enter years: '))
print()

savings = initial_savings
for i in range(years):
    print(f' Savings in year {i}: ${savings:.2f}')
    savings = savings + (savings*interest_rate)

print('\n')

```

range(10, 21, 2)

Starting from 10, the sequence counts up every 2 integers to 20, stopping before 21.

range(5, -6, -1)

Starting from 5, the sequence counts down to -5, stopping before -6.

While vs. for loops

As a general rule:

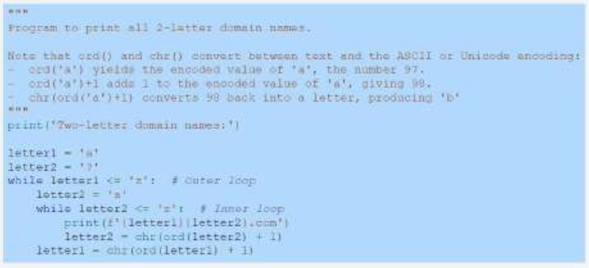
1. *Use a for loop* when the number of iterations is computable before entering the loop, as when counting down from X to 0, printing a string N times, etc.
2. *Use a for loop* when accessing the elements of a container, as when adding 1 to every element in a list, or printing the key of every entry in a dict, etc.
3. *Use a while loop* when the number of iterations is not computable before entering the loop, as when iterating until a user enters a particular character.

Nested loops

A **nested loop** is a loop that appears in the body of another loop. The nested loops are commonly referred to as the **outer loop** and **inner loop**

```
for i in range(2):
    for j in range(3):
        # Inner loop body
```

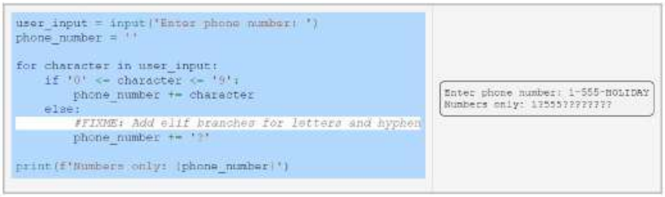
6



```python
"""
Program to print all 2-letter domain names.

Note that ord() and chr() convert between text and the ASCII or Unicode encoding:
-  ord('a') yields the encoded value of 'a', the number 97.
-  ord('a')+1 adds 1 to the encoded value of 'a', giving 98.
-  chr(ord('a')+1) converts 98 back into a letter, producing 'b'
"""
print('Two-letter domain names:')

letter1 = 'a'
letter2 = '?'
while letter1 <= 'z':  # Outer loop
    letter2 = 'a'
    while letter2 <= 'z':  # Inner loop
        print(f'{letter1}{letter2}.com')
        letter2 = chr(ord(letter2) + 1)
    letter1 = chr(ord(letter1) + 1)
```

Incremental development is the process of writing, compiling, and testing a small amount of code, then writing, compiling, and testing a small amount more (an incremental amount) and so on

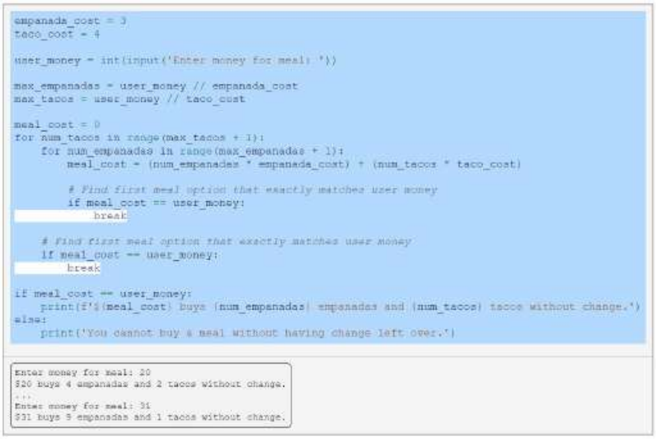
A **FIXME comment** attracts attention to code that needs to be fixed in the future



```
user_input = input('Enter phone number: ')
phone_number = ''

for character in user_input:
    if '0' <= character <= '9':
        phone_number += character
    else:
        #FIXME: Add elif branches for letters and hyphen
        phone_number += '?'

print(f'Numbers only: {phone_number}')
```

```
Enter phone number: 1-555-HOLIDAY
Numbers only: 1?555???????
```

Break and continue

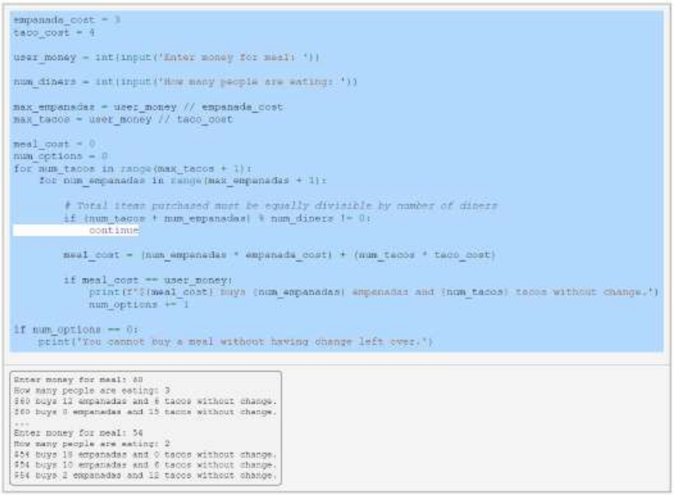


```
empanada_cost = 3
taco_cost = 4

user_money = int(input('Enter money for meal: '))

max_empanadas = user_money // empanada_cost
max_tacos = user_money // taco_cost

meal_cost = 0
for num_tacos in range(max_tacos + 1):
    for num_empanadas in range(max_empanadas + 1):
        meal_cost = (num_empanadas * empanada_cost) + (num_tacos * taco_cost)

        # Find first meal option that exactly matches user money
        if meal_cost == user_money:
            break

    # Find first meal option that exactly matches user money
    if meal_cost == user_money:
        break

if meal_cost == user_money:
    print(f'${meal_cost} buys {num_empanadas} empanadas and {num_tacos} tacos without change.')
else:
    print('You cannot buy a meal without having change left over.')
```

```
Enter money for meal: 20
$20 buys 4 empanadas and 2 tacos without change.
...
Enter money for meal: 31
$31 buys 9 empanadas and 1 tacos without change.
```

A **break** statement in a loop causes the loop to exit immediately - sometimes makes the loop easier to understand

A **continue** statement in a loop causes an immediate jump to the while or for loop header statement - can improve the readability of a loop



```
empanada_cost = 3
taco_cost = 4

user_money = int(input('Enter money for meal: '))

num_diners = int(input('How many people are eating: '))

max_empanadas = user_money // empanada_cost
max_tacos = user_money // taco_cost

meal_cost = 0
num_options = 0
for num_tacos in range(max_tacos + 1):
    for num_empanadas in range(max_empanadas + 1):

        # Total items purchased must be equally divisible by number of diners
        if (num_tacos + num_empanadas) % num_diners != 0:
            continue

        meal_cost = (num_empanadas * empanada_cost) + (num_tacos * taco_cost)

        if meal_cost == user_money:
            print(f'${meal_cost} buys {num_empanadas} empanadas and {num_tacos} tacos without change.')
            num_options += 1

if num_options == 0:
    print('You cannot buy a meal without having change left over.')
```

```
Enter money for meal: 60
How many people are eating: 3
$60 buys 12 empanadas and 6 tacos without change.
$60 buys 0 empanadas and 15 tacos without change.
...
Enter money for meal: 54
How many people are eating: 2
$54 buys 18 empanadas and 0 tacos without change.
$54 buys 10 empanadas and 6 tacos without change.
$54 buys 2 empanadas and 12 tacos without change.
```

The **loop else** construct executes if the loop completes normally and doesn't use a break statement

```
names = ['Janice', 'Clarice', 'Martin', 'Veronica', 'Jason']

num = int(input('Enter number of names to print: '))
for i in range(len(names)):
    if i == num:
        break
    print(names[i], end= ' ')
else:
    print('All names printed.')
```

```
Enter number of names to print: 2
Janice Clarice
...
Enter number of names to print: 8
Janice Clarice Martin Veronica Jason
All names printed.
```

Construct 6.11.1: While loop else.

```
while expression:   # Loop expression
    # Loop body: Sub-statements to execute if
    # the expression evaluated to True
else:
    # Else body: Sub-statements to execute once
    # if the expression evaluated to False

# Statements to execute after the loop
```

Construct 6.11.2: For loop else.

```
for name in iterable:
    # Loop body: Sub-statements to execute
    # for each item in iterable
else:
    # Else body: Sub-statements to execute
    # once when loop completes

# Statements to execute after the loop
```

Getting both index and value when looping

Figure 6.12.1: Using range() and len() to iterate over a sequence.

```
origins = [4, 8, 10]

for index in range(len(origins)):
    value = origins[index]  # Retrieve value of element in list.
    print(f'Element {index}: {value}')
```

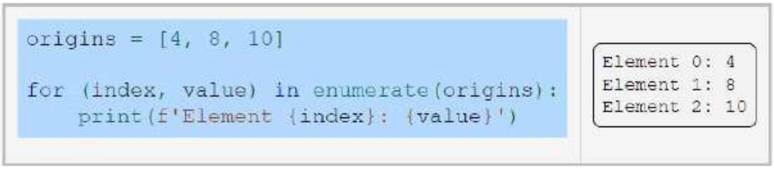
```
Element 0: 4
Element 1: 8
Element 2: 10
```

Figure 6.12.2: Using list.index() to find the index of each element.

```
origins = [4, 8, 10]

for value in origins:
    index = origins.index(value)  # Retrieve index of value in list
    print(f'Element {index}: {value}')
```

```
Element 0: 4
Element 1: 8
Element 2: 10
```

The **enumerate()** function retrieves both the index and corresponding element value at the same time, providing a cleaner and more readable solution



```
origins = [4, 8, 10]

for (index, value) in enumerate(origins):
    print(f'Element {index}: {value}')
```

```
Element 0: 4
Element 1: 8
Element 2: 10
```

The enumerate() function yields a new tuple each iteration of the loop, with the tuple containing the current index and corresponding element value. Unpacking is a process that performs multiple assignments at once, binding comma-separated names on the left to the elements of a sequence on the right

num1, num2 = [350, 400]

Functions

Program redundancy can be reduced by creating a grouping of predefined statements for repeatedly used operations, known as a function

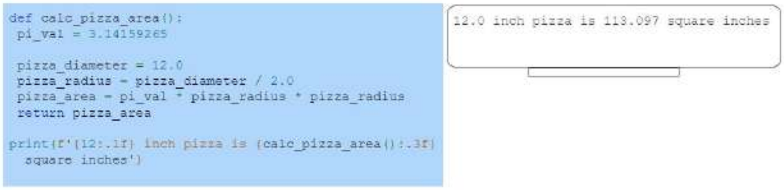
A function is a named series of statements

- A function definition consists of the function's name and a block of statements

  def calc_pizza_area(): followed by an indented block of statements

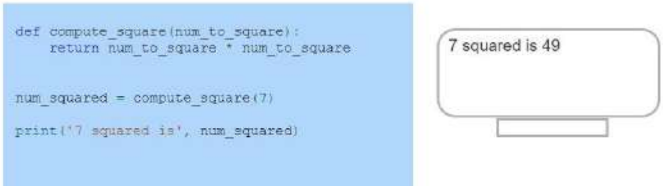
- A function call is an invocation of the function's name, causing the function's statements to execute

The def keyword is used to create new functions



```
def calc_pizza_area():
 pi_val = 3.14159265

 pizza_diameter = 12.0
 pizza_radius = pizza_diameter / 2.0
 pizza_area = pi_val * pizza_radius * pizza_radius
 return pizza_area

print(f'{12:.1f} inch pizza is {calc_pizza_area():.3f}
  square inches')
```

```
12.0 inch pizza is 113.097 square inches
```



```
def compute_square(num_to_square):
    return num_to_square * num_to_square


num_squared = compute_square(7)

print('7 squared is', num_squared)
```

```
7 squared is 49
```

A function may return one value using a **return statement** - not two or more

A function with no return statement, or a return statement with no following expression, returns the value **None** - special keyword that indicates no value

A programmer can influence a function's behavior via an input

- A **parameter** is a function input specified in a function definition

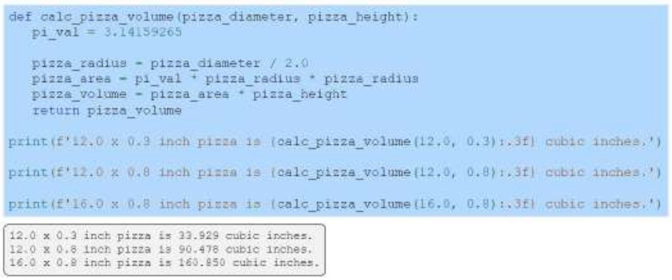
  Ex: A pizza area function might have diameter as an input

- An **argument** is a value provided to a function's parameter during a function call

  Ex: A pizza area function might be called as PrintPizzaArea(12.0) or as PrintPizzaArea(16.0)

A function may have multiple parameters separated by commas or no parameters as in:

```
def calc_sum()
```



```
def calc_pizza_volume(pizza_diameter, pizza_height):
   pi_val = 3.14159265

   pizza_radius = pizza_diameter / 2.0
   pizza_area = pi_val * pizza_radius * pizza_radius
   pizza_volume = pizza_area * pizza_height
   return pizza_volume

print(f'12.0 x 0.3 inch pizza is {calc_pizza_volume(12.0, 0.3):.3f} cubic inches.')

print(f'12.0 x 0.8 inch pizza is {calc_pizza_volume(12.0, 0.8):.3f} cubic inches.')

print(f'16.0 x 0.8 inch pizza is {calc_pizza_volume(16.0, 0.8):.3f} cubic inches.')
```

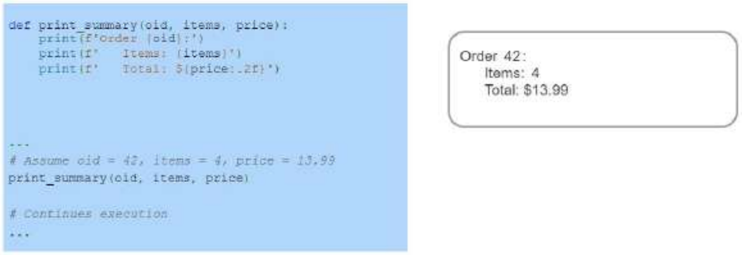
```
12.0 x 0.3 inch pizza is 33.929 cubic inches.
12.0 x 0.8 inch pizza is 90.478 cubic inches.
16.0 x 0.8 inch pizza is 160.850 cubic inches.
```

A function's statements may include function calls, known as **hierarchical function calls** or **nested function calls** as in:

```
user_input = int(input())
```

Print Functions

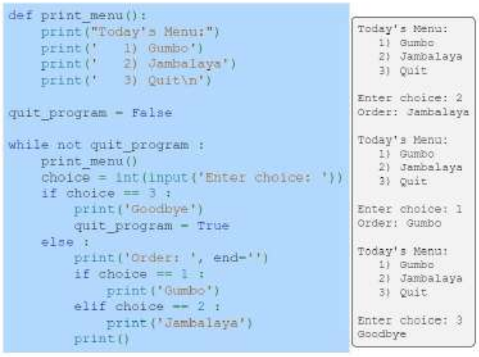
A common operation for a function is to print text. A function that only prints typically does not return a value. A function with no return statement is called a **void function** - returns the value None



```
def print_summary(oid, items, price):
    print(f'Order {oid}:')
    print(f'   Items: {items}')
    print(f'   Total: ${price:.2f}')




...
# Assume oid = 42, items = 4, price = 13.99
print_summary(oid, items, price)

# Continues execution
...
```

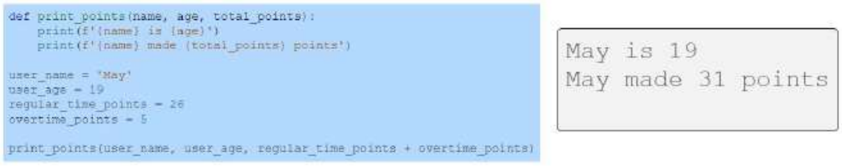
```
Order 42:
    Items: 4
    Total: $13.99
```

Figure 7.2.1: Example: Menu System.



```
def print_menu():
    print("Today's Menu:")
    print('    1) Gumbo')
    print('    2) Jambalaya')
    print('    3) Quit\n')

quit_program = False

while not quit_program :
    print_menu()
    choice = int(input('Enter choice: '))
    if choice == 3 :
        print('Goodbye')
        quit_program = True
    else :
        print('Order: ', end='')
        if choice == 1 :
            print('Gumbo')
        elif choice == 2 :
            print('Jambalaya')
        print()
```

```
Today's Menu:
    1) Gumbo
    2) Jambalaya
    3) Quit

Enter choice: 2
Order: Jambalaya

Today's Menu:
    1) Gumbo
    2) Jambalaya
    3) Quit

Enter choice: 1
Order: Gumbo

Today's Menu:
    1) Gumbo
    2) Jambalaya
    3) Quit

Enter choice: 3
Goodbye
```



```
def print_points(name, age, total_points):
    print(f'{name} is {age}')
    print(f'{name} made {total_points} points')

user_name = 'May'
user_age = 19
regular_time_points = 26
overtime_points = 5

print_points(user_name, user_age, regular_time_points + overtime_points)
```

```
May is 19
May made 31 points
```

Dynamic Typing

Dynamic and static typing

A programmer can pass any type of object as an argument to a function. Consider a function add(x, y) that adds the two parameters:

A programmer can call the `add()` function using two integer arguments, as in `add(5, 7)`, which returns a value of 12. Alternatively, a programmer can pass in two string arguments, as in `add('Tora', 'Bora')`, which would concatenate the two strings and return `'ToraBora'`.

The function's behavior of adding together different types is a concept called polymorphism

(2 * 3) = 6

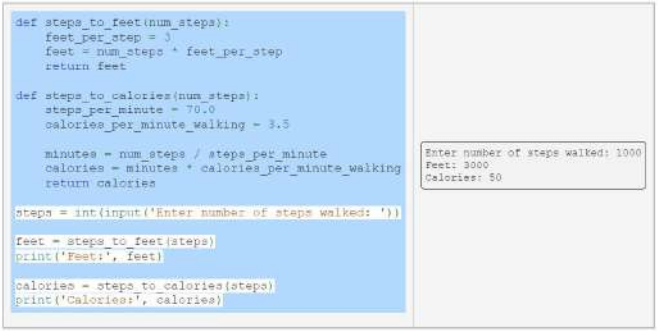
('x' * 3) = 'xxx'

Python uses dynamic typing to determine the type of objects as a program executes

Static typing requires the programmer to define the type of every variable and every function parameter in the code

Reasons for defining functions

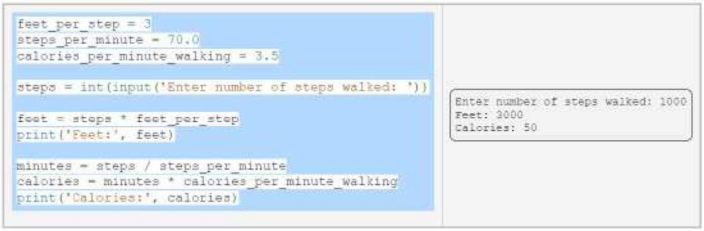
Figure 7.4.1: With user-defined functions, the main program is easy to understand.



```
def steps_to_feet(num_steps):
    feet_per_step = 3
    feet = num_steps * feet_per_step
    return feet

def steps_to_calories(num_steps):
    steps_per_minute = 70.0
    calories_per_minute_walking = 3.5

    minutes = num_steps / steps_per_minute
    calories = minutes * calories_per_minute_walking
    return calories

steps = int(input('Enter number of steps walked: '))

feet = steps_to_feet(steps)
print('Feet:', feet)

calories = steps_to_calories(steps)
print('Calories:', calories)
```

```
Enter number of steps walked: 1000
Feet: 3000
Calories: 50
```

Figure 7.4.2: Without user-defined functions, the main program is harder to read and understand.



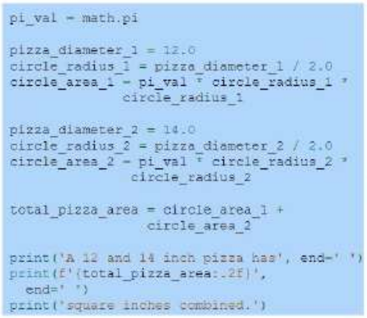
```
feet_per_step = 3
steps_per_minute = 70.0
calories_per_minute_walking = 3.5

steps = int(input('Enter number of steps walked: '))

feet = steps * feet_per_step
print('Feet:', feet)

minutes = steps / steps_per_minute
calories = minutes * calories_per_minute_walking
print('Calories:', calories)
```

```
Enter number of steps walked: 1000
Feet: 3000
Calories: 50
```

Programmers commonly use functions to write programs modularly and incrementally

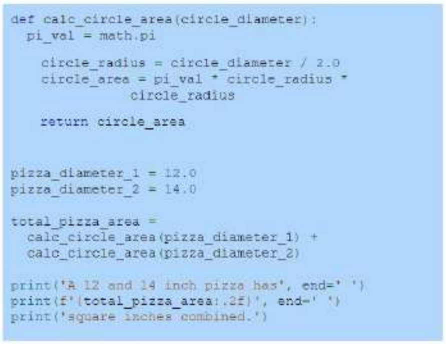
- **Modular development** is the process of dividing a program into seperate modules that can be developed and tested separately and then integrated into a single program

- **Incremental development** is a process in which a programmer writes and tests a few statements, then writes and tests a small amount more, and so on



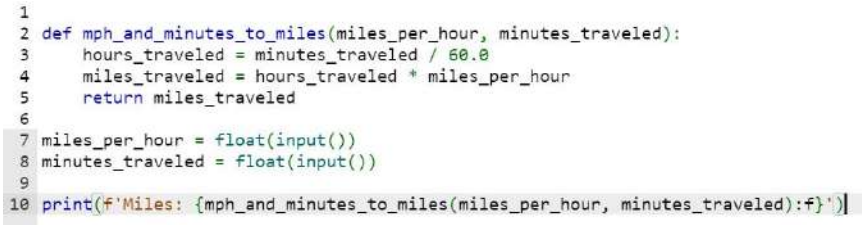
```
pi_val = math.pi

pizza_diameter_1 = 12.0
circle_radius_1 = pizza_diameter_1 / 2.0
circle_area_1 = pi_val * circle_radius_1 *
                circle_radius_1

pizza_diameter_2 = 14.0
circle_radius_2 = pizza_diameter_2 / 2.0
circle_area_2 = pi_val * circle_radius_2 *
                circle_radius_2

total_pizza_area = circle_area_1 +
                   circle_area_2

print('A 12 and 14 inch pizza has', end=' ')
print(f'{total_pizza_area:.2f}',
  end=' ')
print('square inches combined.')
```

main program with redundant code

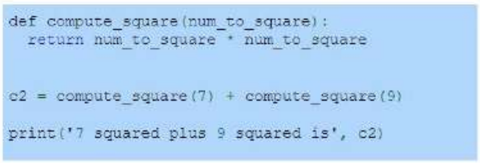


```
def calc_circle_area(circle_diameter):
  pi_val = math.pi

    circle_radius = circle_diameter / 2.0
    circle_area = pi_val * circle_radius *
                  circle_radius

    return circle_area


pizza_diameter_1 = 12.0
pizza_diameter_2 = 14.0

total_pizza_area =
  calc_circle_area(pizza_diameter_1) +
  calc_circle_area(pizza_diameter_2)

print('A 12 and 14 inch pizza has', end=' ')
print(f'{total_pizza_area:.2f}', end=' ')
print('square inches combined.')
```

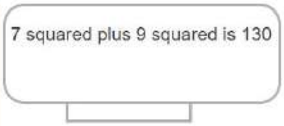
main program calls calc_circle_area() avoiding redundant code



```

def mph_and_minutes_to_miles(miles_per_hour, minutes_traveled):
    hours_traveled = minutes_traveled / 60.0
    miles_traveled = hours_traveled * miles_per_hour
    return miles_traveled

miles_per_hour = float(input())
minutes_traveled = float(input())

print(f'Miles: {mph_and_minutes_to_miles(miles_per_hour, minutes_traveled):f}')
```

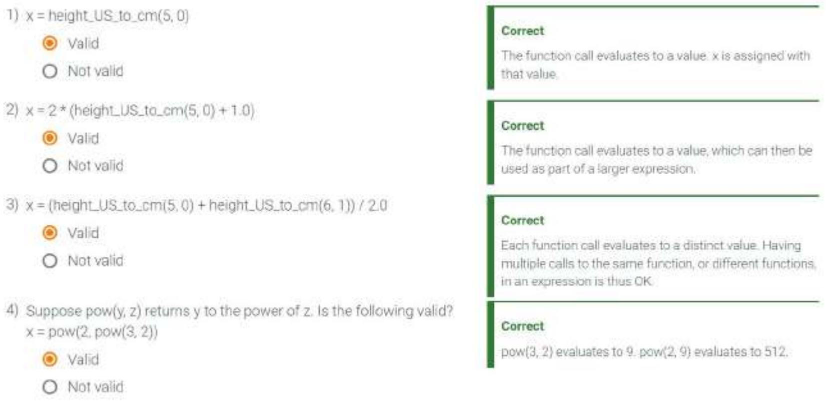


```
def compute_square(num_to_square):
  return num_to_square * num_to_square


c2 = compute_square(7) + compute_square(9)

print('7 squared plus 9 squared is', c2)
```



```
7 squared plus 9 squared is 130
```

The expression then evaluates to c2 = 49 + 81, which assigns variable c2 with 130. Lastly, the print statement executes.



1) x = height_US_to_cm(5, 0)

[✓ Valid] Not valid

Correct

The function call evaluates to a value. x is assigned with that value.

2) x = 2 * (height_US_to_cm(5, 0) + 1.0)

[✓ Valid] Not valid

Correct

The function call evaluates to a value, which can then be used as part of a larger expression.

3) x = (height_US_to_cm(5, 0) + height_US_to_cm(6, 1)) / 2.0

[✓ Valid] Not valid

Correct

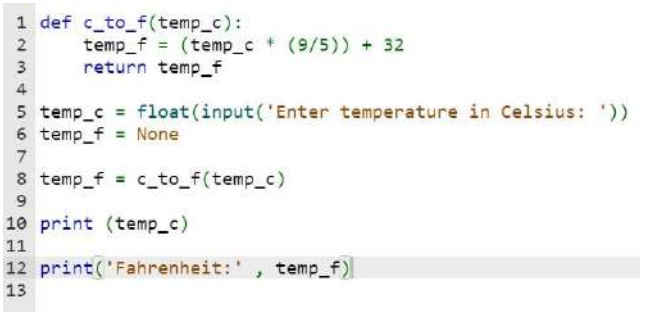
Each function call evaluates to a distinct value. Having multiple calls to the same function, or different functions, in an expression is thus OK.

4) Suppose pow(y, z) returns y to the power of z. Is the following valid?
x = pow(2, pow(3, 2))

[✓ Valid] Not valid

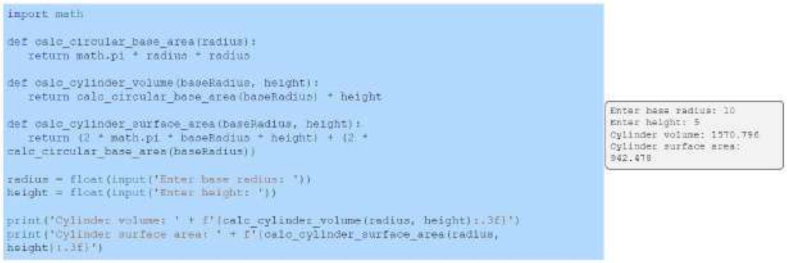
Correct

pow(3, 2) evaluates to 9. pow(2, 9) evaluates to 512.



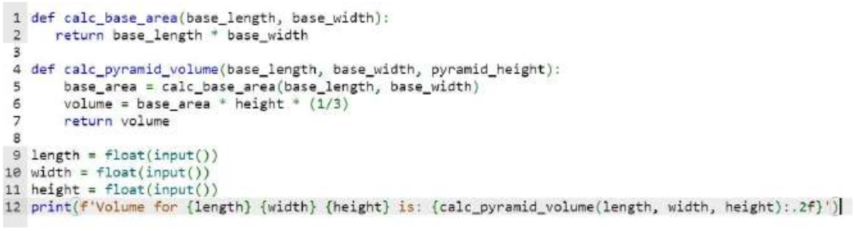
```python
def c_to_f(temp_c):
    temp_f = (temp_c * (9/5)) + 32
    return temp_f

temp_c = float(input('Enter temperature in Celsius: '))
temp_f = None

temp_f = c_to_f(temp_c)

print (temp_c)

print('Fahrenheit:' , temp_f)

```

Figure 7.5.1: Program that calculates cylinder volume and surface area by calling a modular function for the cylinder's base.



```python
import math

def calc_circular_base_area(radius):
   return math.pi * radius * radius

def calc_cylinder_volume(baseRadius, height):
   return calc_circular_base_area(baseRadius) * height

def calc_cylinder_surface_area(baseRadius, height):
   return (2 * math.pi * baseRadius * height) + (2 *
calc_circular_base_area(baseRadius))

radius = float(input('Enter base radius: '))
height = float(input('Enter height: '))

print('Cylinder volume: ' + f'{calc_cylinder_volume(radius, height):.3f}')
print('Cylinder surface area: ' + f'{calc_cylinder_surface_area(radius,
height):.3f}')
```

```
Enter base radius: 10
Enter height: 5
Cylinder volume: 1570.796
Cylinder surface area:
942.478
```



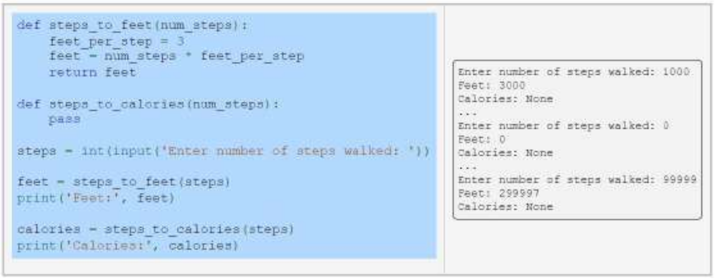
```python
def calc_base_area(base_length, base_width):
   return base_length * base_width

def calc_pyramid_volume(base_length, base_width, pyramid_height):
    base_area = calc_base_area(base_length, base_width)
    volume = base_area * height * (1/3)
    return volume

length = float(input())
width = float(input())
height = float(input())
print(f'Volume for {length} {width} {height} is: {calc_pyramid_volume(length, width, height):.2f}')
```

Function Stubs

Programs are typically written using incremental development - meaning a programmer writes and tests a few statements, then writes and tests a small amount more, and so on

To assist with the incremental development process, programers commonly introduce function stubs, which are function definitions whose statements have not yet been written yet

A programmer should consider whether or not calling the unwritten function is a valid operation. One approach is to use the pass keyword, which performs no operation except to act as a place-holder for a required statement

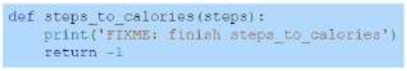


```
def steps_to_feet(num_steps):
    feet_per_step = 3
    feet = num_steps * feet_per_step
    return feet

def steps_to_calories(num_steps):
    pass

steps = int(input('Enter number of steps walked: '))

feet = steps_to_feet(steps)
print('Feet:', feet)

calories = steps_to_calories(steps)
print('Calories:', calories)
```

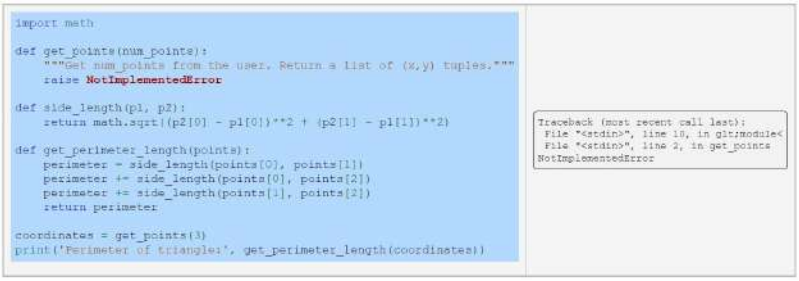
```
Enter number of steps walked: 1000
Feet: 3000
Calories: None
...
Enter number of steps walked: 0
Feet: 0
Calories: None
...
Enter number of steps walked: 99999
Feet: 299997
Calories: None
```

Another approach is to print a message when a function stub is called - good practice is for a stub to return -1 for a function that will have a return value

Figure 7.6.2: A function stub using a print statement.



```
def steps_to_calories(steps):
    print('FIXME: finish steps_to_calories')
    return -1
```

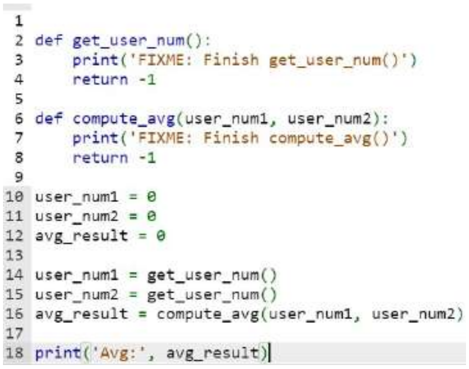


```
import math

def get_points(num_points):
    """Get num_points from the user. Return a list of (x,y) tuples."""
    raise NotImplementedError

def side_length(p1, p2):
    return math.sqrt((p2[0] - p1[0])**2 + (p2[1] - p1[1])**2)

def get_perimeter_length(points):
    perimeter = side_length(points[0], points[1])
    perimeter += side_length(points[0], points[2])
    perimeter += side_length(points[1], points[2])
    return perimeter

coordinates = get_points(3)
print('Perimeter of triangle:', get_perimeter_length(coordinates))
```

```
Traceback (most recent call last):
 File "<stdin>", line 10, in glt;module<
 File "<stdin>", line 2, in get_points
NotImplementedError
```

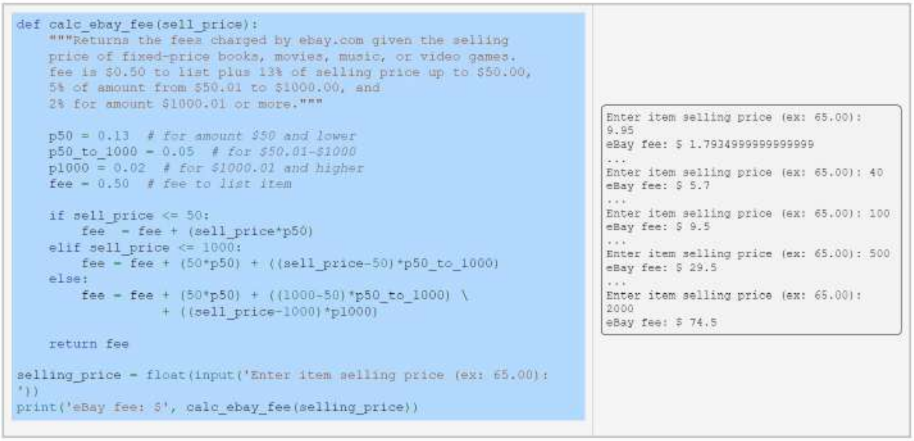
To stop executing if an unfinished function is called an NotImplementedError can be generated with the statement

raise NotImplementedError



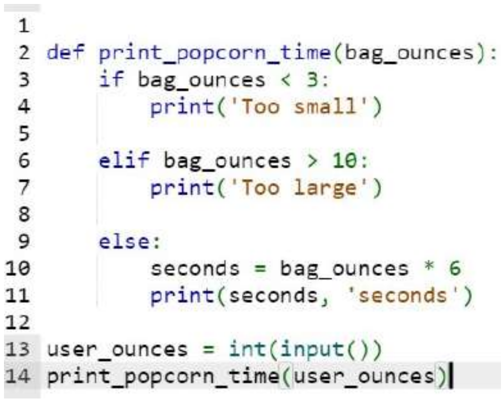
```python

def get_user_num():
    print('FIXME: Finish get_user_num()')
    return -1

def compute_avg(user_num1, user_num2):
    print('FIXME: Finish compute_avg()')
    return -1

user_num1 = 0
user_num2 = 0
avg_result = 0

user_num1 = get_user_num()
user_num2 = get_user_num()
avg_result = compute_avg(user_num1, user_num2)

print('Avg:', avg_result)
```

Functions with Branches/Loops



```python
def calc_ebay_fee(sell_price):
    """Returns the fees charged by ebay.com given the selling
    price of fixed-price books, movies, music, or video games.
    fee is $0.50 to list plus 13% of selling price up to $50.00,
    5% of amount from $50.01 to $1000.00, and
    2% for amount $1000.01 or more."""

    p50 = 0.13  # for amount $50 and lower
    p50_to_1000 = 0.05  # for $50.01-$1000
    p1000 = 0.02  # for $1000.01 and higher
    fee = 0.50  # fee to list item

    if sell_price <= 50:
        fee  = fee + (sell_price*p50)
    elif sell_price <= 1000:
        fee = fee + (50*p50) + ((sell_price-50)*p50_to_1000)
    else:
        fee = fee + (50*p50) + ((1000-50)*p50_to_1000) \
                  + ((sell_price-1000)*p1000)

    return fee

selling_price = float(input('Enter item selling price (ex: 65.00):
'))
print('eBay fee: $', calc_ebay_fee(selling_price))
```

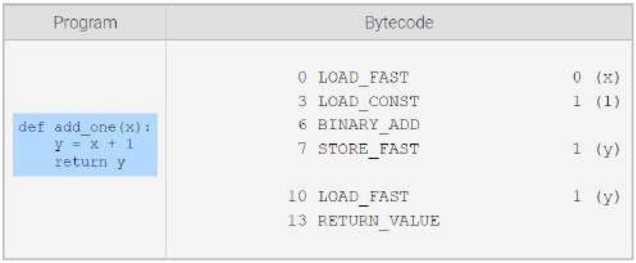
```
Enter item selling price (ex: 65.00):
9.95
eBay fee: $ 1.7934999999999999
...
Enter item selling price (ex: 65.00): 40
eBay fee: $ 5.7
...
Enter item selling price (ex: 65.00): 100
eBay fee: $ 9.5
...
Enter item selling price (ex: 65.00): 500
eBay fee: $ 29.5
...
Enter item selling price (ex: 65.00):
2000
eBay fee: $ 74.5
```



```python

def print_popcorn_time(bag_ounces):
    if bag_ounces < 3:
        print('Too small')

    elif bag_ounces > 10:
        print('Too large')

    else:
        seconds = bag_ounces * 6
        print(seconds, 'seconds')

user_ounces = int(input())
print_popcorn_time(user_ounces)
```

Functions are Objects

A function definition like def print_face(): creates a new function object with the name print_face bound to that object

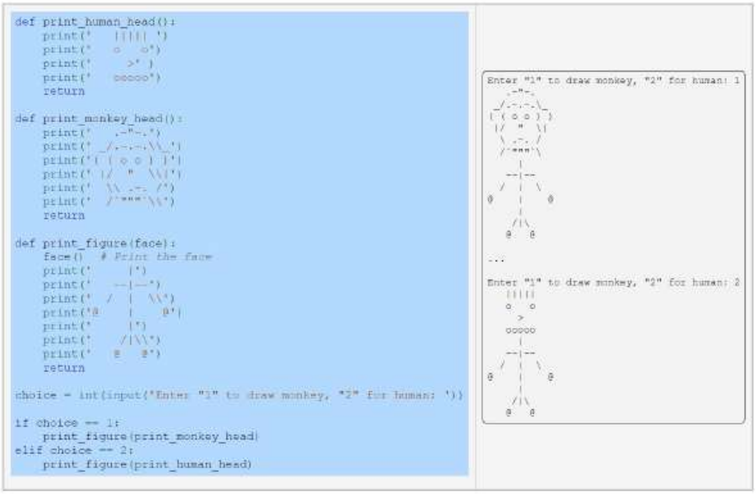


Program	Bytecode

`def add_one(x):`
`    y = x + 1`
`    return y`
	`0 LOAD_FAST 0 (x)`
	`3 LOAD_CONST 1 (1)`
	`6 BINARY_ADD`
	`7 STORE_FAST 1 (y)`
	
	`10 LOAD_FAST 1 (y)`
	`13 RETURN_VALUE`

A part of the value of a function object is compiled bytecode that represents the statements to be executed by the function

Figure 7.8.2: Functions can be passed as arguments.



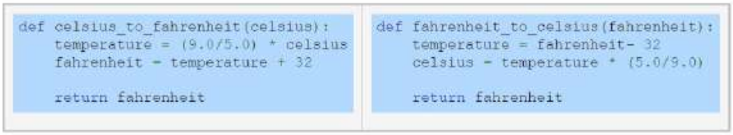
```
def print_human_head():
    print('    |||||  ')
    print('    o   o')
    print('      >' )
    print('    ooooo')
    return

def print_monkey_head():
    print('    .-"-.')
    print('  _/.-.-.\\_')
    print(' ( ( o o ) )')
    print('  |/  "  \\|')
    print('   \\ .-. /')
    print('   /`"""`\\')
    return

def print_figure(face):
    face()  # Print the face
    print('      |')
    print('    --|--')
    print('   /  |  \\')
    print('@     |     @')
    print('      |')
    print('     /|\\')
    print('    @   @')
    return

choice = int(input('Enter "1" to draw monkey, "2" for human: '))

if choice == 1:
    print_figure(print_monkey_head)
elif choice == 2:
    print_figure(print_human_head)
```

```
Enter "1" to draw monkey, "2" for human: 1
    .-"-.
  _/.-.-.\_
 ( ( o o ) )
  |/  "  \|
   \ .-. /
   /`"""`\
      |
    --|--
   /  |  \
@     |     @
      |
     /|\
    @   @

...

Enter "1" to draw monkey, "2" for human: 2
    |||||
    o   o
      >
    ooooo
      |
    --|--
   /  |  \
@     |     @
      |
     /|\
    @   @
```

Copy-paste errors

A common error is to copy-and-paste code among functions but then not complete all necessary modifications to the pasted code. For example, a programmer might have developed and tested a function to convert a temperature value in Celsius to Fahrenheit, and then copied and modified the original function into a new function to convert Fahrenheit to Celsius as shown:

Figure 7.9.1: Copy-paste common error: Pasted code not properly modified. Find error on the right.



```
def celsius_to_fahrenheit(celsius):
    temperature = (9.0/5.0) * celsius
    fahrenheit = temperature + 32

    return fahrenheit
```

```
def fahrenheit_to_celsius(fahrenheit):
    temperature = fahrenheit- 32
    celsius = temperature * (5.0/9.0)

    return fahrenheit
```

Return errors

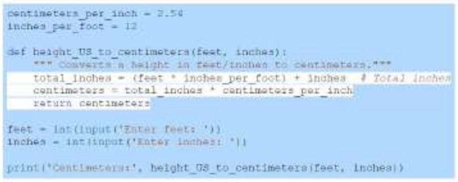
Another common error is to return the wrong variable, like if `return temperature` *had been used in the temperature conversion program by accident.* The function will work and sometimes even return the correct value.

Another common error is to fail to return a value for a function. If execution reaches the end of a function's statements without encountering a return statement, then the function returns a value of `None`. If the function is expected to return an actual value, then such an assignment can cause confusion.

Scope of Variables and Functions

A variable or function object is only visible to part of a program, known as the object's scope - when a variable is created inside a function, the variable's scope is limited to inside that function

Figure 7.10.1: Variable scope.



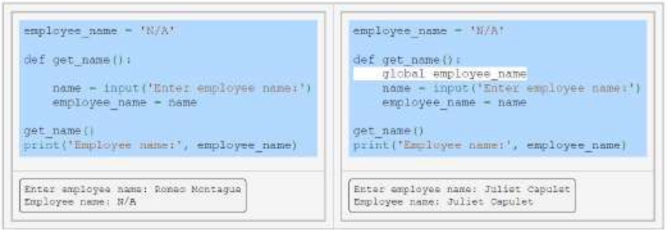
```
centimeters_per_inch = 2.54
inches_per_foot = 12

def height_US_to_centimeters(feet, inches):
    """ Converts a height in feet/inches to centimeters."""
    total_inches = (feet * inches_per_foot) + inches  # Total inches
    centimeters = total_inches * centimeters_per_inch
    return centimeters

feet = int(input('Enter feet: '))
inches = int(input('Enter inches: '))

print('Centimeters:', height_US_to_centimeters(feet, inches))
```

Local variable scope extends from assignment to end of function. Global variable scope extends to end of file.

The function's variables total_inches and centimeters are invisible to the code outside of the function and cannot be used. Such variables defined inside a function are called local variables

In contrast, a variable defined outside of a function is called a global variable

Figure 7.10.2: The global statement (right) allows modifying a global variable.



```
employee_name = 'N/A'

def get_name():

    name = input('Enter employee name:')
    employee_name = name

get_name()
print('Employee name:', employee_name)
```

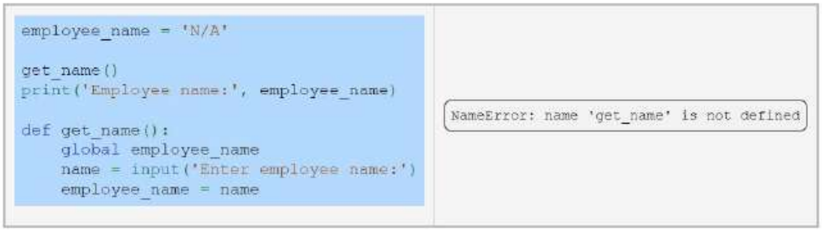
```
Enter employee name: Romeo Montague
Employee name: N/A
```

```
employee_name = 'N/A'

def get_name():
    global employee_name
    name = input('Enter employee name:')
    employee_name = name

get_name()
print('Employee name:', employee_name)
```

```
Enter employee name: Juliet Capulet
Employee name: Juliet Capulet
```

A global statement must be used to change the value of a global variable inside of a function

Figure 7.10.3: Function definitions must be evaluated before that function is called.



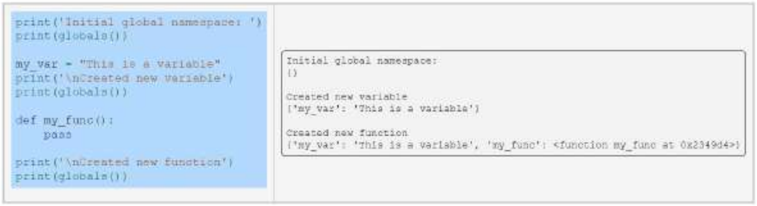
```
employee_name = 'N/A'

get_name()
print('Employee name:', employee_name)

def get_name():
    global employee_name
    name = input('Enter employee name:')
    employee_name = name
```

```
NameError: name 'get_name' is not defined
```

Namespaces and Scope Resolution

A namespace maps names to objects

Figure 7.11.1: Using the globals() to get namespace names.



```
print('Initial global namespace: ')
print(globals())

my_var = "This is a variable"
print('\nCreated new variable')
print(globals())

def my_func():
    pass

print('\nCreated new function')
print(globals())
```

```
Initial global namespace:
{}

Created new variable
{'my_var': 'This is a variable'}

Created new function
{'my_var': 'This is a variable', 'my_func': <function my_func at 0x2349d4>}
```

Scope and scope resolution

Scope is the area of code where a name is visible. Namespaces are used to make scope work. Each scope, such as global scope or a local function scope, has its own namespace. If a namespace contains a name at a specific location in the code, then that name is visible and a programmer can use it in an expression.

There are at least three nested scopes that are active at any point in a program's execution: [1]

1. Built-in scope – Contains all of the built-in names of Python, such as `int()`, `str()`, `list()`, `range()`, etc.
2. Global scope – Contains all globally defined names outside of any functions.
3. Local scope – Usually refers to scope within the currently executing function, but is the same as global scope if no function is executing.

When a name is referenced in code, the local scope's namespace is the first checked, followed by the global scope, and finally the built-in scope. If the name cannot be found in any namespace, the interpreter generates a `NameError`. The process of searching for a name in the available namespaces is called ***scope resolution***.

Function Arguments

Arguments to functions are passed by object reference, a concept known in Python as pass-by-assignment. When a function is called, new local variables are created in the function's local namespace by binding the names in the parameter list to the passed arguments

The semantics of passing object references as arguments is important because modifying an argument that is referenced elsewhere in the program may cause side effects outside of the function scope. When a function modifies a parameter, whether or not that modification is seen outside the scope of the function depends on the *mutability* of the argument object.

- If the object is **immutable**, such as a string or integer, then the modification is limited to inside the function. Any modification to an immutable object results in the creation of a *new* object in the function's local scope, thus leaving the original argument object unchanged.
- If the object is **mutable**, then in-place modification of the object can be seen outside the scope of the function. Any operation like adding elements to a container or sorting a list that is performed within a function will also affect any other variables in the program that reference the same object.

The following program illustrates how the modification of a list argument's elements inside a function persists outside of the function call.

Keyword Arguments and Default Parameter Values

A function with many arguments can become very long and difficult to read

```
def print_book_description(title, author, publisher, year, version, num_chapters, num_pages):
    # Format and print description of a book...

print_book_description('The Lord of the Rings', 'J. R. R. Tolkien', 'George Allen & Unwin',
                       1954, 1.0, 22, 456)
```

Python provides for **keyword arguments** that allow arguments to map to parameters by name, instead of implicitly by position in the argument list

```
def print_book_description(title, author, publisher, year, version, num_chapters, num_pages):
    # Format and print description of a book...

print_book_description(title='The Lord of the Rings', publisher='George Allen & Unwin',
                       year=1954, author='J. R. R. Tolkien', version=1.0,
                       num_pages=456, num_chapters=22)
```

Keyword arguments provide a bit of clarity to potentially confusing function calls. *Good practice is to use keyword arguments for any function containing more than approximately 4 arguments.*

Keyword arguments can be mixed with positional arguments, provided that the keyword arguments come last. *A common error is to place keyword arguments before all position arguments, which generates an exception.*

Figure 7.13.3: All keyword arguments must follow positional arguments.

```
def split_check(amount, num_people, tax_percentage, tip_percentage):
    # ...

split_check(125.00, tip_percentage=0.15, num_people=2, tax_percentage=0.095)
```

Sometimes a function has parameters that are optional. A function can have a **default parameter value** for one or more parameters, meaning that a function call can optionally omit an argument, and the default parameter value will be substituted for the corresponding omitted argument

```
def print_date(day, month, year, style=0):
    if style == 0:  # American
        print(month, '/', day, '/', year)
    elif style == 1:  # European
        print(day, '/', month, '/', year)
    else:
        print('Invalid Style')

print_date(30, 7, 2012, 0)
print_date(30, 7, 2012, 1)
print_date(30, 7, 2012)  # style argument not provided! Default value of 0 used.
```

```
7 / 30 / 2012
30 / 7 / 2012
7 / 30 / 2012
```

A parameter's **default value** is the value used in the absence of an argument in the function call

```
print_date(30, 7, 2012, 0)    # Defaults: none
print_date(30, 7, 2012)       # Defaults:                               style=0
print_date(30, 7)             # Defaults:                   year=2000, style=0
print_date(30)                # Defaults:          month=1, year=2000, style=0
print_date()                  # Defaults: day=1, month=1, year=2000, style=0
```

If a parameter does not have a default value, then failing to provide an argument (either keyword or positional) generates an error.

A common error is to provide a mutable object, like a list, as a default parameter. Such a definition can be problematic because the default argument object is created only once, at the time the function is defined (when the script is loaded), and not every time the function is called. Modification of the default parameter object will persist across function calls, which is likely not what a programmer intended. The below program demonstrates the problem with mutable default objects and illustrates a solution that creates a new empty list each time the function is called.

Figure 7.13.6: Mutable default objects remain changed over multiple function calls.

Default object modification

```
def append_to_list(value, my_list=[]):
    my_list.append(value)
    return my_list

numbers = append_to_list(50)   # default list
appended with 50
print(numbers)
numbers = append_to_list(100)   # default list
appended with 100
print(numbers)
```

```
[50]
[50, 100]
```

Solution: Make new list

```
def append_to_list(value, my_list=None):   # Use default
parameter value of None
    if my_list == None:   # Create a new list if a list
was not provided
        my_list = []

    my_list.append(value)
    return my_list

numbers = append_to_list(50)   # default list appended
with 50
print(numbers)
numbers = append_to_list(100)   # default list appended
with 100
print(numbers)
```

```
[50]
[100]
```

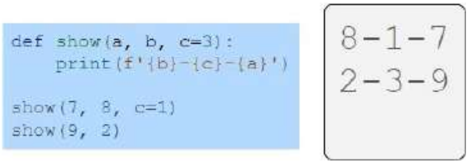
Mixing keyword arguments and default parameter values

Mixing keyword arguments and default parameter values allows a programmer to omit arbitrary arguments from a function call. Because keyword arguments use names instead of position to match arguments to parameters, any argument can be omitted as long as that argument has a default value.

Consider the `print_date` function from above. If every parameter has a default value, then the user can use keyword arguments to pass specific arguments anywhere in the argument list. Below are some sample function calls:

Figure 7.13.7: Mixing keyword arguments and default parameter values allows omitting arbitrary arguments.

```
def print_date(day=1, month=1, year=2000, style=0):
    # ...


print_date(day=30, year=2012)    # Defaults:        month=1,             style=0
print_date(style=1)              # Defaults: day=1, month=1, year=2000
print_date(year=2012, month=4)   # Defaults: day=1,                      style=0
```

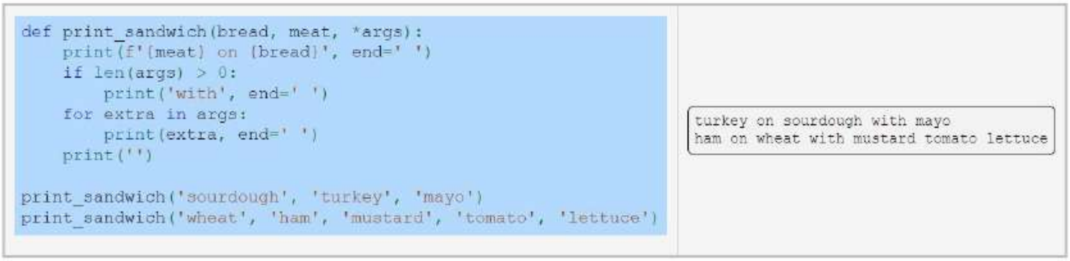


```
def show(a, b, c=3):
    print(f'{b}-{c}-{a}')

show(7, 8, c=1)
show(9, 2)
```

```
8-1-7
2-3-9
```

Arbitrary Argument Lists

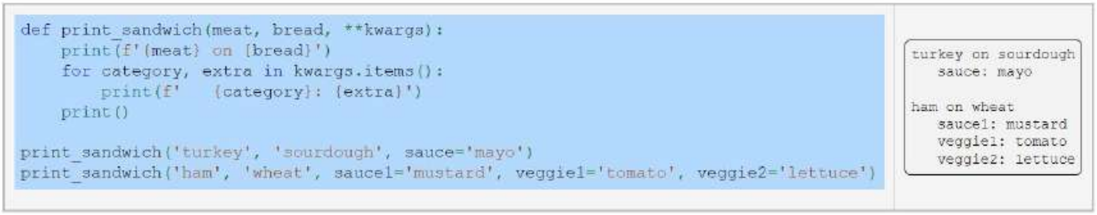
Sometimes a programmer doesn't know how many arguments a function requires. A function definition can include a *args parameter that collects optional positional parameters into an arbitrary argument list tuple



```
def print_sandwich(bread, meat, *args):
    print(f'{meat} on {bread}', end=' ')
    if len(args) > 0:
        print('with', end=' ')
    for extra in args:
        print(extra, end=' ')
    print('')

print_sandwich('sourdough', 'turkey', 'mayo')
print_sandwich('wheat', 'ham', 'mustard', 'tomato', 'lettuce')
```

```
turkey on sourdough with mayo
ham on wheat with mustard tomato lettuce
```

Adding a final function parameter of **Kwargs creates a dictionary containing "extra" arguments not defined in the function definition; Kwargs is short for keyword arguments. The keys of the dictionary are the parameter names specified in the function call



```
def print_sandwich(meat, bread, **kwargs):
    print(f'{meat} on {bread}')
    for category, extra in kwargs.items():
        print(f'   {category}: {extra}')
    print()

print_sandwich('turkey', 'sourdough', sauce='mayo')
print_sandwich('ham', 'wheat', sauce1='mustard', veggie1='tomato', veggie2='lettuce')
```

```
turkey on sourdough
   sauce: mayo

ham on wheat
   sauce1: mustard
   veggie1: tomato
   veggie2: lettuce
```

The * and ** characters in *args and **kwargs are the important symbols. Using "args" and "kwargs" is standard practice, but any valid identifier is acceptable (like perhaps using *condiments in the sandwich example).

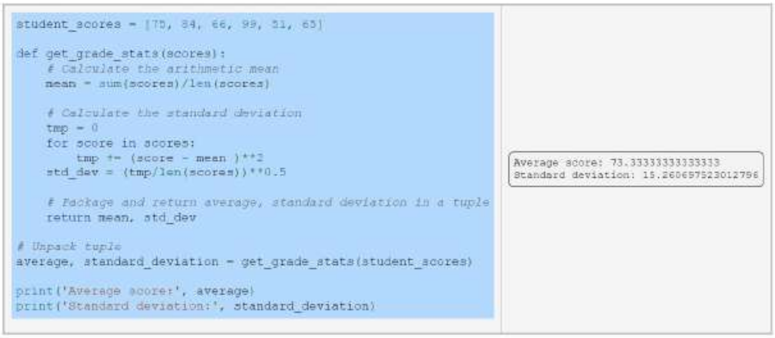
One or both of *args or **kwargs can be used. They must come last (and in that order if both are used) in the parameter list, otherwise an error occurs.

Multiple Function Outputs

Multiple function outputs

Occasionally a function should produce multiple output values. However, function return statements are limited to returning only one value. A workaround is to package the multiple outputs into a single container, commonly a tuple, and to then return that container.

Figure 7.15.1: Multiple outputs can be returned in a container.



```
student_scores = [75, 84, 66, 99, 51, 65]

def get_grade_stats(scores):
    # Calculate the arithmetic mean
    mean = sum(scores)/len(scores)

    # Calculate the standard deviation
    tmp = 0
    for score in scores:
        tmp += (score - mean )**2
    std_dev = (tmp/len(scores))**0.5

    # Package and return average, standard deviation in a tuple
    return mean, std_dev

# Unpack tuple
average, standard_deviation = get_grade_stats(student_scores)

print('Average score:', average)
print('Standard deviation:', standard_deviation)
```

```
Average score: 73.33333333333333
Standard deviation: 15.260697523012796
```

The statement average, standard_deviation = get_grade_stats(student_scores) utilizes unpacking to perform multiple assignments at once, so that average and standard_deviation are assigned the first and second elements from the returned tuple

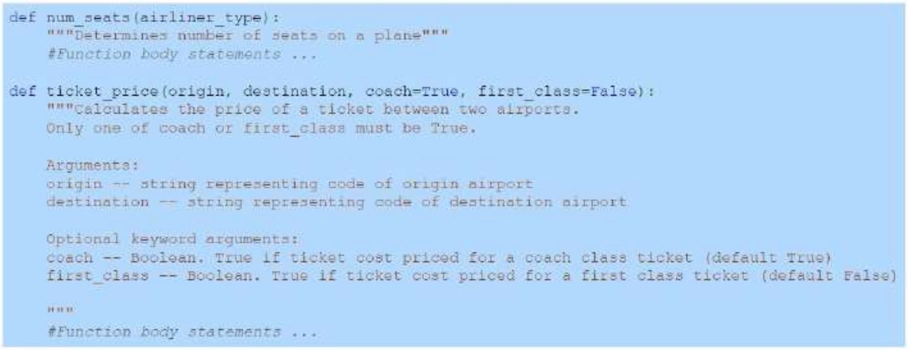
Docstrings

A large program can contain many functions with a wide variety of uses. A programmer should document each function, giving a high-level description of the purpose of the function, so that later readers of the code can more easily understand. A ***docstring*** is a string literal placed in the first line of a function body.

A docstring starts and ends with three consecutive quotation marks. *Good practice is to keep the docstring of a simple function as a single line, including the quotes. Furthermore, there should be no blank lines before or after the docstring.*

Multi-line docstrings can be used for more complicated functions to describe the function arguments. Multi-line docstrings should use consistent indentation for each line, separating the ending triple-quotes by a blank line.

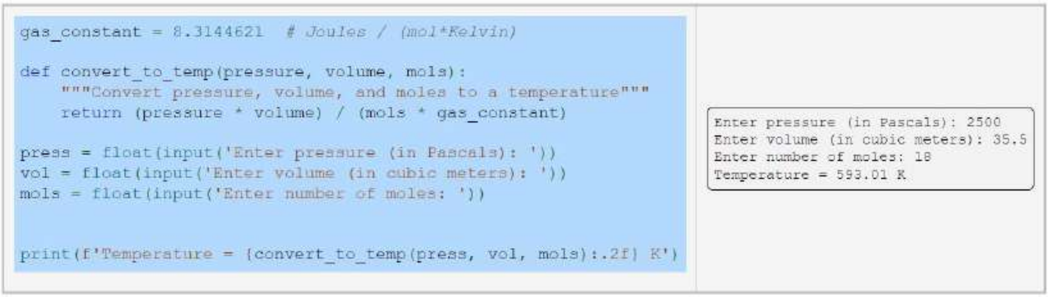
Figure 7.16.1: A single and a multi-line docstring.



```
def num_seats(airliner_type):
    """Determines number of seats on a plane"""
    #Function body statements ...

def ticket_price(origin, destination, coach=True, first_class=False):
    """Calculates the price of a ticket between two airports.
    Only one of coach or first_class must be True.

    Arguments:
    origin -- string representing code of origin airport
    destination -- string representing code of destination airport

    Optional keyword arguments:
    coach -- Boolean. True if ticket cost priced for a coach class ticket (default True)
    first_class -- Boolean. True if ticket cost priced for a first class ticket (default False)

    """
    #Function body statements ...
```

The help() function can aid a programmer by providing them with all the documentation associated with an object. A statement such as help(ticket_price) would print out the docstring for the ticket_price() function, providing the programmer with information about how to call that function

Figure 7.17.1: PV = nRT. Compute the temperature of a gas.



```
gas_constant = 8.3144621  # Joules / (mol*Kelvin)

def convert_to_temp(pressure, volume, mols):
    """Convert pressure, volume, and moles to a temperature"""
    return (pressure * volume) / (mols * gas_constant)

press = float(input('Enter pressure (in Pascals): '))
vol = float(input('Enter volume (in cubic meters): '))
mols = float(input('Enter number of moles: '))


print(f'Temperature = {convert_to_temp(press, vol, mols):.2f} K')
```

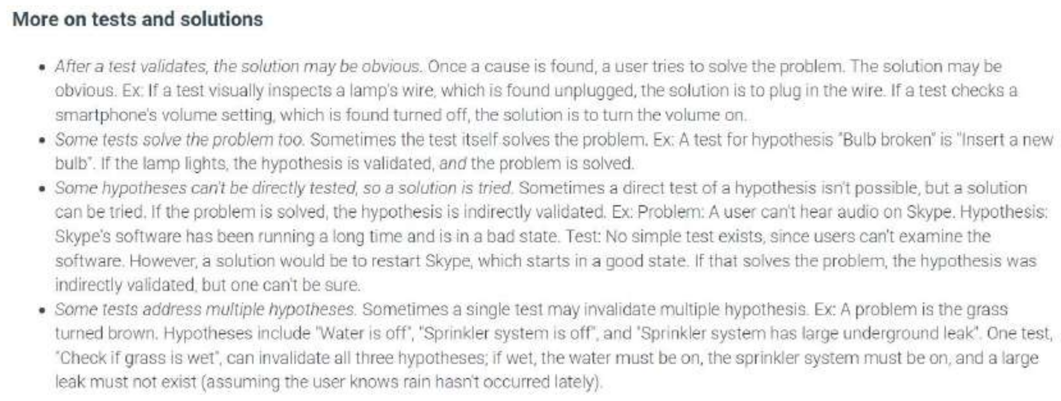
```
Enter pressure (in Pascals): 2500
Enter volume (in cubic meters): 35.5
Enter number of moles: 18
Temperature = 593.01 K
```

Troubleshooting and Debugging

Troubleshooting is a systematic process for finding and fixing a problem's cause in a system

When troubleshooting a problem, a hypothesis should be a statement of a possible cause, stated so as to be either true or false

Hierarchical hypotheses a hypothesis may be decomposed into more precise sub-hypotheses



More on tests and solutions

- *After a test validates, the solution may be obvious.* Once a cause is found, a user tries to solve the problem. The solution may be obvious. Ex: If a test visually inspects a lamp's wire, which is found unplugged, the solution is to plug in the wire. If a test checks a smartphone's volume setting, which is found turned off, the solution is to turn the volume on.
- *Some tests solve the problem too.* Sometimes the test itself solves the problem. Ex: A test for hypothesis "Bulb broken" is "Insert a new bulb". If the lamp lights, the hypothesis is validated, *and* the problem is solved.
- *Some hypotheses can't be directly tested, so a solution is tried.* Sometimes a direct test of a hypothesis isn't possible, but a solution can be tried. If the problem is solved, the hypothesis is indirectly validated. Ex: Problem: A user can't hear audio on Skype. Hypothesis: Skype's software has been running a long time and is in a bad state. Test: No simple test exists, since users can't examine the software. However, a solution would be to restart Skype, which starts in a good state. If that solves the problem, the hypothesis was indirectly validated, but one can't be sure.
- *Some tests address multiple hypotheses.* Sometimes a single test may invalidate multiple hypothesis. Ex: A problem is the grass turned brown. Hypotheses include "Water is off", "Sprinkler system is off", and "Sprinkler system has large underground leak". One test, "Check if grass is wet", can invalidate all three hypotheses; if wet, the water must be on, the sprinkler system must be on, and a large leak must not exist (assuming the user knows rain hasn't occurred lately).

Binary search divides an item into two halves, runs a test to decide in which half something lies, and repeats the binary search on that half

A program is a series of instructions (statements) a computer executes to perform a calculation

In a program, a problem's cause is called a bug, and troubleshooting is called debugging

A basic debugging process is visual inspection - looking at each statement one-by-one to try to find a bug

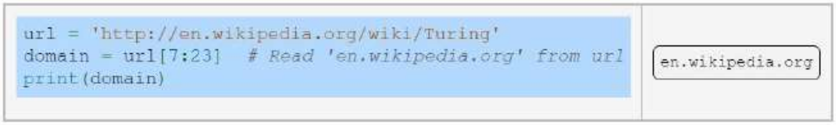
Another debugging process is to insert debug output statements whose output helps determine whether the preceding statement has the bug

Strings

String Slicing

Strings are a sequence type, having characters ordered by index from left to right. An **index** is an integer matching to a specific position in a string's sequence of characters

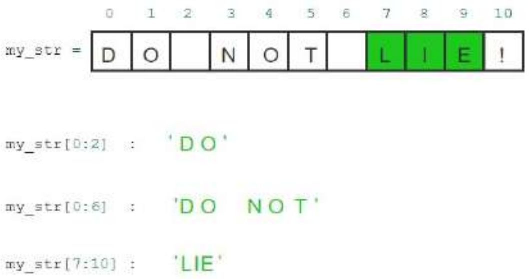
A programmer often needs to read more than one character at a time. Multiple consecutive characters can be read using slice notation



```
url = 'http://en.wikipedia.org/wiki/Turing'
domain = url[7:23]   # Read 'en.wikipedia.org' from url
print(domain)
```

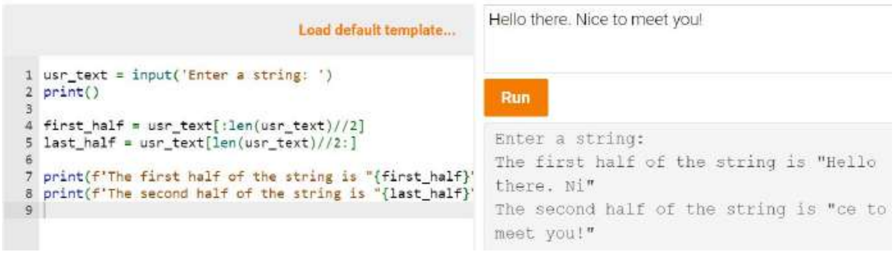
```
en.wikipedia.org
```

Slice notation has the form my_str[start:end], which creates a new string whose value contains the characters of my_str from indices start to end -1



Variables can be used in place of literals to specify slice notation start and end indices

my_str[x:y]

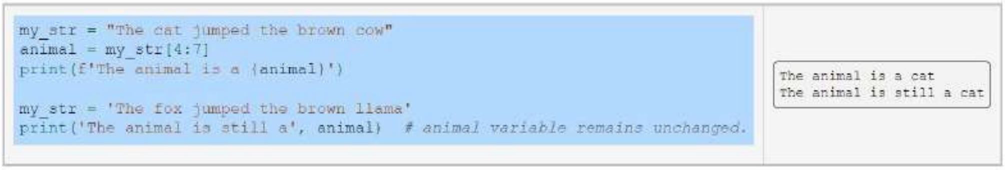


```
usr_text = input('Enter a string: ')
print()

first_half = usr_text[:len(usr_text)//2]
last_half = usr_text[len(usr_text)//2:]

print(f'The first half of the string is "{first_half}'
print(f'The second half of the string is "{last_half}'
```

```
Hello there. Nice to meet you!
```

```
Enter a string:
The first half of the string is "Hello
there. Ni"
The second half of the string is "ce to
meet you!"
```

Figure 10.1.2: A slice creates a new object.



```
my_str = "The cat jumped the brown cow"
animal = my_str[4:7]
print(f'The animal is a {animal}')

my_str = 'The fox jumped the brown llama'
print('The animal is still a', animal)  # animal variable remains unchanged.
```

```
The animal is a cat
The animal is still a cat
```

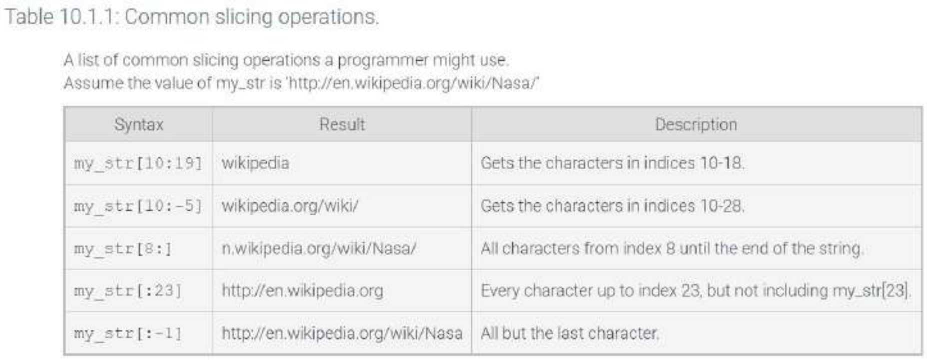


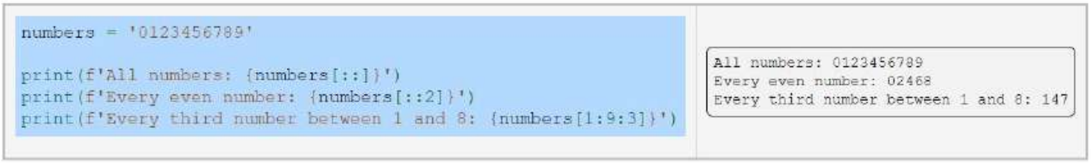
Table 10.1.1: Common slicing operations.

A list of common slicing operations a programmer might use.
Assume the value of my_str is 'http://en.wikipedia.org/wiki/Nasa/'

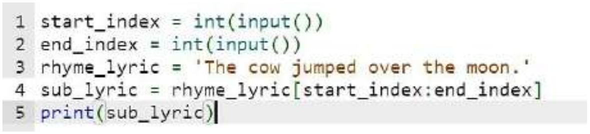
Syntax	Result	Description
`my_str[10:19]`	wikipedia	Gets the characters in indices 10-18.
`my_str[10:-5]`	wikipedia.org/wiki/	Gets the characters in indices 10-28.
`my_str[8:]`	n.wikipedia.org/wiki/Nasa/	All characters from index 8 until the end of the string.
`my_str[:23]`	http://en.wikipedia.org	Every character up to index 23, but not including my_str[23].
`my_str[:-1]`	http://en.wikipedia.org/wiki/Nasa	All but the last character.

Slice notation also provides for a third argument, known as the stride. The stride determines how much to increment the index after each element

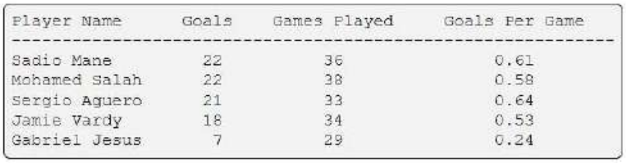
my_str[0:10:2] reads every other element between 0 and 10. The stride defaults to 1 if not specified


```
numbers = '0123456789'

print(f'All numbers: {numbers[::]}')
print(f'Every even number: {numbers[::2]}')
print(f'Every third number between 1 and 8: {numbers[1:9:3]}')
```

```
All numbers: 0123456789
Every even number: 02468
Every third number between 1 and 8: 147
```


```
start_index = int(input())
end_index = int(input())
rhyme_lyric = 'The cow jumped over the moon.'
sub_lyric = rhyme_lyric[start_index:end_index]
print(sub_lyric)
```

Advanced String Formatting

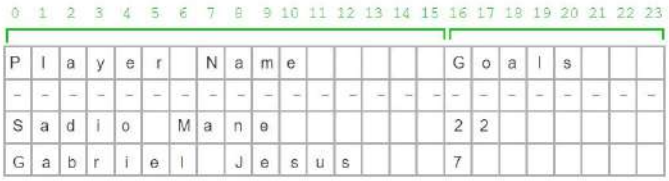

```
Player Name        Goals    Games Played     Goals Per Game
-----------------------------------------------------------------
Sadio Mane           22          36               0.61
Mohamed Salah        22          38               0.58
Sergio Aguero        21          33               0.64
Jamie Vardy          18          34               0.53
Gabriel Jesus         7          29               0.24
```

Note in the above example how the text is formatted into columns with the contents of each column (except the leftmost column) centered under the column header. A programmer could achieve this careful formatting by placing spaces into their output strings, but each row would require different numbers of spaces depending on the player name (longer names require fewer spaces between the first and second columns).

A format specification may include a **field width** that defines the minimum number of characters that must be inserted into the string

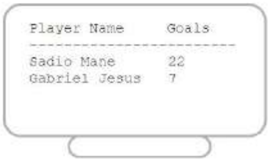
{name:16} specifies a width of 16 characters

```
print(f'{"Player Name":16}{"Goals":8}')
print('-' * 24)

print(f'{"Sadio Mane":16}{"22":8}')
print(f'{"Gabriel Jesus":16}{"7":8}')
```



Strings are left-aligned

Numbers are right-aligned



```
Player Name      Goals
------------------------
Sadio Mane       22
Gabriel Jesus    7
```

A format specification can include an **alignment character** that determines how a value should be aligned within the width of the field

```
names = ['Sadio Mane', 'Gabriel Jesus']
goals = [22, 7]

print(<f-string 1>)        #Replaced in table below
print('-' * 24)
for i in range(2):
    print(<f-string 2>)    #Replaced in table below
```

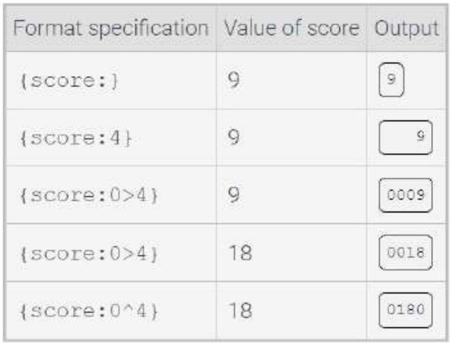
Alignment type	`<f-string 1>`
	`<f-string 2>`
		Output

Left-aligned	`f'{"Player Name":<16}{"Goals":<8}'`
	`f'{names[i]:<16}{goals[i]:<8}'`
		`Player Name      Goals`
		`------------------------`
		`Sadio Mane       22`
		`Gabriel Jesus    7`

Right-aligned	`f'{"Player Name":>16}{"Goals":>8}'`
	`f'{names[i]:>16}{goals[i]:>8}'`
		`     Player Name   Goals`
		`------------------------`
		`      Sadio Mane      22`
		`   Gabriel Jesus       7`

Centered	`f'{"Player Name":^16}{"Goals":^8}'`
	`f'{names[i]:^16}{goals[i]:^8}'`
		`  Player Name    Goals`
		`------------------------`
		`   Sadio Mane      22`
		` Gabriel Jesus     7`

The basic set of possible alignment options include left-aligned '<', right-aligned '>', and centered '^'



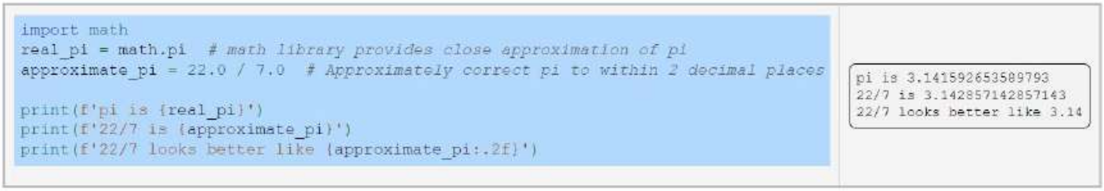
Format specification	Value of score	Output
{score:}	9	9
{score:4}	9	9
{score:0>4}	9	0009
{score:0>4}	18	0018
{score:0^4}	18	0180

The fill character is used to pad a replacement field when the string being inserted is smaller than the field width

{score:0>4} is 0009 if score is 9

A programmer commonly wants to set how many digits to the right of a floating-point number to print. The optional precision component of a format specification indicates how many digits to the right of the decimal should be included in the output of floating types

f'{1.725:.1f}' would print 1.7



```
import math
real_pi = math.pi  # math library provides close approximation of pi
approximate_pi = 22.0 / 7.0  # Approximately correct pi to within 2 decimal places

print(f'pi is {real_pi}')
print(f'22/7 is {approximate_pi}')
print(f'22/7 looks better like {approximate_pi:.2f}')
```

```
pi is 3.141592653589793
22/7 is 3.142857142857143
22/7 looks better like 3.14
```

String Methods

Finding and replacing

A common task for a programmer is to edit the contents of a string. Recall that string objects are immutable -- once created, strings can not be changed. To update a string variable, a new string object must be created and bound to the variable name, replacing the old object. The *replace* string method provides a simple way to create a new string by replacing all occurrences of a substring with a new substring.

- ***replace(old, new)*** -- Returns a copy of the string with all occurrences of the substring old replaced by the string new. The old and new arguments may be string variables or string literals.
- ***replace(old, new, count)*** -- Same as above, except only replaces the first count occurrences of old.

Some methods are useful for finding the position of where a character or substring is located in a string:

- ***find(x)*** -- Returns the index of the first occurrence of item x in the string, else returns -1. x may be a string variable or string literal. Recall that in a string, the index of the first character is 0, not 1. If `my_str` is 'Boo Hoo!':
    - `my_str.find('!')  # Returns 7`
    - `my_str.find('Boo')  # Returns 0`
    - `my_str.find('oo')  # Returns 1 (first occurrence only)`
- ***find(x, start)*** -- Same as find(x), but begins the search at index start:
    - `my_str.find('oo', 2)  # Returns 5`
- ***find(x, start, end)*** -- Same as find(x, start), but stops the search at index end - 1:
    - `my_str.find('oo', 2, 4)  # Returns -1 (not found)`
- ***rfind(x)*** -- Same as find(x) but searches the string in reverse, returning the last occurrence in the string.

Another useful function is count, which counts the number of times a substring occurs in the string:

- ***count(x)*** -- Returns the number of times x occurs in the string.
    - `my_str.count('oo')  # Returns 2`

Note that methods such as `find()` and `rfind()` are useful only for cases where a programmer needs to know the exact location of the character or substring in the string. If the exact position is not important, then the `in` membership operator should be used to check if a character or substring is contained in the string:

```
if 'batman' in superhero_name:
    # Statements to execute if superhero_name contains 'batman' in any position.
```

String objects may be compared using relational operators (<, <=, >, >=), equality operators (==, !=), membership operators (in, not in), and identity operators (is, is not)

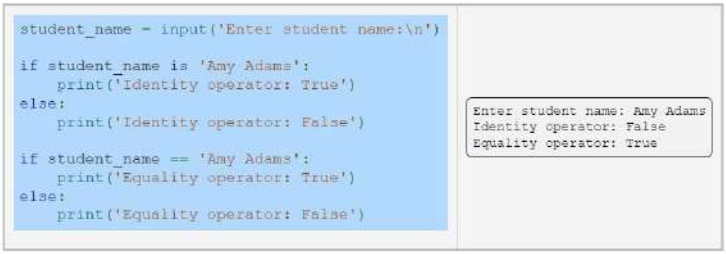
Example	Expression result	Why?
`'Hello' == 'Hello'`	True	The strings are exactly identical values
`'Hello' == 'Hello!'`	False	The left hand string does not end with '!'
`'Yankee Sierra' > 'Amy Wise'`	True	The first character of the left side 'Y' is "greater than" (in ASCII value) the first character of the right side 'A'
`'Yankee Sierra' > 'Yankee Zulu'`	False	The characters of both sides match until the second word. The first character of the second word on the left 'S' is not "greater than" (in ASCII value) the first character on the right side 'Z'
`'seph' in 'Joseph'`	True	The substring 'seph' can be found starting at the 3rd position of 'Joseph'
`'jo' in 'Joseph'`	False	'jo' (with a lowercase 'j') is not in 'Joseph' (with an uppercase 'J')

If one string is shorter than the other with all corresponding characters equal, then the shorter string is considered less than the longer string.

The membership operators (`in`, `not in`) provide a simple method for detecting whether a specific substring exists in the string. The argument to the right of the operator is examined for the existence of the argument on the left. Note that reversing the arguments does not work, as 'Jo' is a substring of 'Kay, Jo', but 'Kay, Jo' is not a substring of 'Jo'.

The identity operators (`is`, `is not`) determine whether the two arguments are bound to the same object. *A common error is to use an identity operator in place of an equality operator.* Ex: A programmer may write `name is 'Amy Adams'`, intending to check if the value of `name` is the same as the literal 'Amy Adams'. Instead, the Python interpreter creates a new string object from the string literal on the right, and compares the identity of the new object to the `name` object, which returns False. *Good practice is to always use the equality operator == when comparing values.*

Figure 10.3.2: Identity vs. equality operators.



```
student_name = input('Enter student name:\n')

if student_name is 'Amy Adams':
    print('Identity operator: True')
else:
    print('Identity operator: False')

if student_name == 'Amy Adams':
    print('Equality operator: True')
else:
    print('Equality operator: False')
```

```
Enter student name: Amy Adams
Identity operator: False
Equality operator: True
```

A number of methods are available to help manage string comparisons. The list below describes the most commonly used methods; a full list is available at docs.python.org.

- Methods to check a string value that returns a True or False Boolean value:
  - ***isalnum()*** -- Returns True if all characters in the string are lowercase or uppercase letters, or the numbers 0-9.
  - ***isdigit()*** -- Returns True if all characters are the numbers 0-9.
  - ***islower()*** -- Returns True if all cased characters are lowercase letters.
  - ***isupper()*** -- Return True if all cased characters are uppercase letters.
  - ***isspace()*** -- Return True if all characters are whitespace.
  - ***startswith(x)*** -- Return True if the string starts with x.
  - ***endswith(x)*** -- Return True if the string ends with x.

Note that the methods `islower()` and `isupper()` ignore non-cased characters. Ex: `'abc?'.islower()` returns True, ignoring the question mark.

Creating new strings from a string

A programmer often needs to transform two strings into similar formats to perform a comparison. The list below shows some of the more common string methods that create string copies, altering the case or amount of whitespace of the original string:

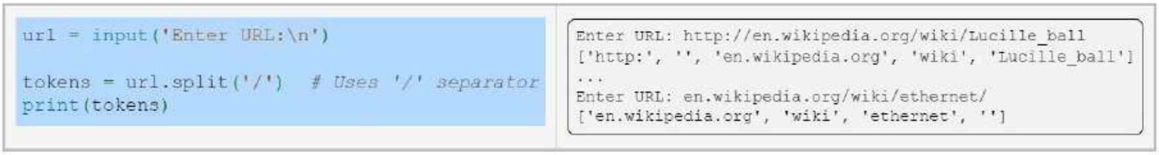
- Methods to create new strings:
  - ***capitalize()*** -- Returns a copy of the string with the first character capitalized and the rest lowercased.
  - ***lower()*** -- Returns a copy of the string with all characters lowercased.
  - ***upper()*** -- Returns a copy of the string with all characters uppercased.
  - ***strip()*** -- Returns a copy of the string with leading and trailing whitespace removed.
  - ***title()*** -- Returns a copy of the string as a title, with first letters of words capitalized.

A user may enter any one of the non-equivalent values 'Bob' , 'BOB ', or 'bob' into a program that reads in names. The statement `name = input().strip().lower()` reads in the user input, strips the leading and trailing whitespace, and changes all the characters to lowercase. Thus, user input of 'Bob', 'BOB ', or 'bob' would each result in name having just the value 'bob'.

Good practice when reading user-entered strings is to apply transformations when reading in data (such as input), as opposed to later in the program. Applying transformations immediately limits the likelihood of introducing bugs because the user entered an unexpected string value. Of course, there are many examples of programs in which capitalization or whitespace should indicate a unique string -- the programmer should use discretion depending on the program being implemented.

Splitting and Joining Strings

A common programming task is to break a large string down into the comprising substrings. The string method split() splits a string into a list of tokens. Each token is a substring that forms a part of a larger string. A separator is a character or sequence of characters that indicates where to split the string into tokens

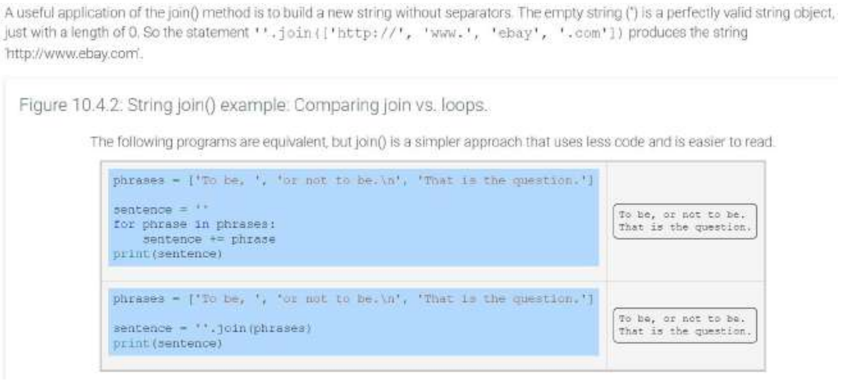


```
url = input('Enter URL:\n')

tokens = url.split('/')  # Uses '/' separator
print(tokens)
```

```
Enter URL: http://en.wikipedia.org/wiki/Lucille_ball
['http:', '', 'en.wikipedia.org', 'wiki', 'Lucille_ball']
...
Enter URL: en.wikipedia.org/wiki/ethernet/
['en.wikipedia.org', 'wiki', 'ethernet', '']
```

If the split string starts or ends with the separator, or if two consecutive separators exist, then the resulting list will contain an empty string for each such occurrence

The join() string method performs the inverse operation of split() by joining a list of strings together to create a single string



A useful application of the join() method is to build a new string without separators. The empty string (') is a perfectly valid string object, just with a length of 0. So the statement `''.join(['http://', 'www.', 'ebay', '.com'])` produces the string 'http://www.ebay.com'.

Figure 10.4.2: String join() example: Comparing join vs. loops.

The following programs are equivalent, but join() is a simpler approach that uses less code and is easier to read.

```
phrases = ['To be, ', 'or not to be.\n', 'That is the question.']

sentence = ''
for phrase in phrases:
    sentence += phrase
print(sentence)
```

```
To be, or not to be.
That is the question.
```

```
phrases = ['To be, ', 'or not to be.\n', 'That is the question.']

sentence = ''.join(phrases)
print(sentence)
```

```
To be, or not to be.
That is the question.
```

my_str = '@'.join(['billgates', 'microsoft']) assigns my_str with the string 'billgates@microsoft'. The separator '@' provides a join() method that accepts a single list argument

Using the split() and join() methods together

The split() and join() methods are commonly used together to replace or remove specific sections of a string. Ex: A programmer may want to change 'C:/Users/Brian/report.txt' to 'C:\\Users\\Brian\\report.txt', perhaps because a different operating system uses different separators to specify file locations. The example below illustrates how split() and join() are used together.

Figure 10.4.3: Splitting and joining: Replacing separators.

```python
path = input('Enter file name: ')

new_separator = input('Enter new separator: ')
tokens = path.split('/')
print(new_separator.join(tokens))
```

```
Enter file name: C:/Users/Wolfman/Documents/report.pdf
Enter new separator: \\
C:\\Users\\Wolfman\\Documents\\report.pdf
```

A programmer may also want to add, remove, or replace specific token(s) from a string. Ex: The program below reads in a URL and checks whether the fourth token (index 3) is 'wiki', as Wikipedia URLs follow the format of http://language.wikipedia.org/wiki/topic. If 'wiki' is missing from the URL, the program uses the list method insert() (explained further elsewhere) to correct the URL by adding 'wiki' before index 3.

Figure 10.4.4: Splitting and joining: Editing tokens.

```python
url = input('Enter Wikipedia URL: ')

tokens = url.split('/')

if 'wiki' != tokens[3]:
    tokens.insert(3, 'wiki')
    new_url = '/'.join(tokens)

    print(f'{url} is not a valid address.')
    print(f'Redirecting to {new_url}')
else:
    print(f'Loading {url}')
```

```
Enter Wikipedia URL: http://en.wikipedia.org/wiki/Rome
Loading http://en.wikipedia.org/wiki/Rome
...
Enter Wikipedia URL: http://en.wikipedia.org/Rome
http://en.wikipedia.org/Rome is not a valid address.
Redirecting to http://en.wikipedia.org/wiki/Rome
```

Lists and Dictionaries

The list object type is one of the most important and often used types in a Python program

A list is a container, which is an object that groups related objects together

Each element in a list can be a different type such as strings, integers, floats, or even other lists

The list() function accepts a single iterable object argument, such as a string, list, or tuple, and returns a new list object

An index is a zero-based integer matching to a specific position in the list's sequence of elements

- Cannot be a floating-point type

```
animals = ['cat', 'dog', 'bird', 'raptor']
print(animals[0])
```

1) `print(animals[0])`

```
cat
```

Table 11.1.1: Some common list operations.

Operation	Description	Example code	Example output

my_list = [1, 2, 3]	Creates a list.	`my_list = [1, 2, 3]`
		`print(my_list)`
			[1, 2, 3]

list(iter)	Creates a list.	`my_list = list('123')`
		`print(my_list)`
			['1', '2', '3']

my_list[index]	Get an element from a list.	`my_list = [1, 2, 3]`
		`print(my_list[1])`
			2

my_list[start:end]	Get a *new* list containing some of another list's elements.	`my_list = [1, 2, 3]`
		`print(my_list[1:3])`
			[2, 3]

my_list1 + my_list2	Get a *new* list with elements of my_list2 added to end of my_list1.	`my_list = [1, 2] + [3]`
		`print(my_list)`
			[1, 2, 3]

my_list[i] = x	Change the value of the ith element in-place.	`my_list = [1, 2, 3]`
		`my_list[2] = 9`
		`print(my_list)`
			[1, 2, 9]

my_list[len(my_list):] = [x]	Add the elements in [x] to the end of my_list. The append(x) method (explained in another section) may be preferred for clarity.	`my_list = [1, 2, 3]`
		`my_list[len(my_list):] = [9]`
		`print(my_list)`
			[1, 2, 3, 9]

del my_list[i]	Delete an element from a list.	`my_list = [1, 2, 3]`
		`del my_list[1]`
		`print(my_list)`
			[1, 3]

A list is mutable and is thus able to grow and shrink without the program having to replace the entire list with an updated copy

Such growing and shrinking capability is called in-place modification

One simple method to create a copy:

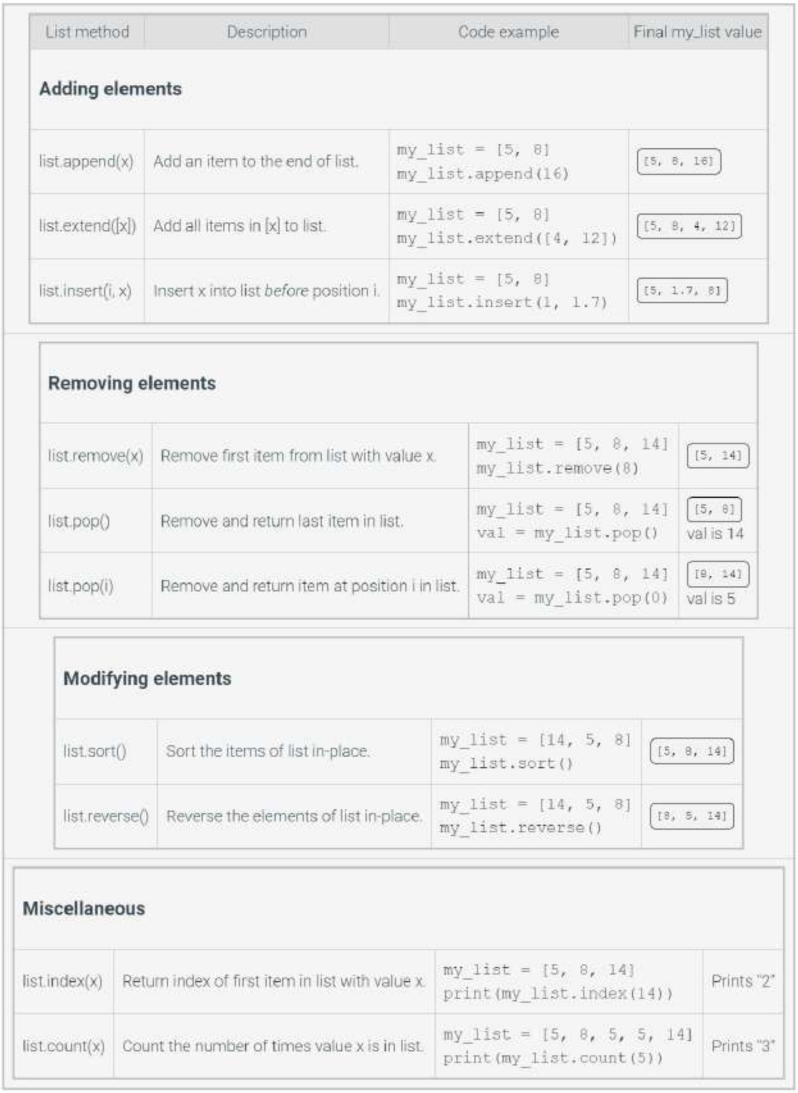
your_teams = my_teams[:]

```
actors = ['Pitt', 'Damon']
actors.insert(1, 'Affleck')
print(actors[0], actors[1], actors[2])
```

```
Pitt Affleck Damon
```

List Methods

A list method can perform a useful operation on a list such as adding or removing elements, sorting, reversing, etc.



List method	Description	Code example	Final my_list value
Adding elements			

list.append(x)	Add an item to the end of list.	`my_list = [5, 8]`
		`my_list.append(16)`
			`[5, 8, 16]`

list.extend([x])	Add all items in [x] to list.	`my_list = [5, 8]`
		`my_list.extend([4, 12])`
			`[5, 8, 4, 12]`

list.insert(i, x)	Insert x into list *before* position i.	`my_list = [5, 8]`
		`my_list.insert(1, 1.7)`
			`[5, 1.7, 8]`

Removing elements			

list.remove(x)	Remove first item from list with value x.	`my_list = [5, 8, 14]`
		`my_list.remove(8)`
			`[5, 14]`

list.pop()	Remove and return last item in list.	`my_list = [5, 8, 14]`
		`val = my_list.pop()`
			`[5, 8]`
			val is 14

list.pop(i)	Remove and return item at position i in list.	`my_list = [5, 8, 14]`
		`val = my_list.pop(0)`
			`[8, 14]`
			val is 5

Modifying elements			

list.sort()	Sort the items of list in-place.	`my_list = [14, 5, 8]`
		`my_list.sort()`
			`[5, 8, 14]`

list.reverse()	Reverse the elements of list in-place.	`my_list = [14, 5, 8]`
		`my_list.reverse()`
			`[8, 5, 14]`

Miscellaneous			

list.index(x)	Return index of first item in list with value x.	`my_list = [5, 8, 14]`
		`print(my_list.index(14))`
			Prints "2"

list.count(x)	Count the number of times value x is in list.	`my_list = [5, 8, 5, 5, 14]`
		`print(my_list.count(5))`
			Prints "3"

1. vals is a list containing elements 1, 4, and 16.
2. The statement vals.append(9) appends element 9 to the end of the list.
3. The statement vals.insert(2, 18) inserts element 18 into position 2 of the list.
4. The statement vals.pop() removes the last element, 9, from the list.
5. The statement vals.remove(4) removes the first instance of element 4 from the list.
6. The statement vals.remove(55) removes the first instance of element 55 from the list. The list does not contain the element 55 so vals is the same.

Iterating over a List

A programmer commonly wants to access each element of a list. Looping through a sequence such as a list is so common that Python supports a construct called a for loop, specifically for iteration purposes

```
for my_var in my_list:
    # Loop body statements go here
```

Figure 11.3.2: Iterating through a list example: Finding the maximum even number.

```
user_input = input('Enter numbers:')

tokens = user_input.split()  # Split into separate strings

# Convert strings to integers
nums = []
for token in tokens:
    nums.append(int(token))

# Print each position and number
print()  # Print a single newline
for index in range(len(nums)):
    value = nums[index]

    print(f'{index}: {value}')

# Determine maximum even number
max_num = None
for num in nums:
    if (max_num == None) and (num % 2 == 0):
        # First even number found
        max_num = num
    elif (max_num != None) and (num > max_num ) and (num % 2 == 0):
        # Larger even number found
        max_num = num

print('Max even #:', max_num)
```

```
Enter numbers:3 5 23 -1 456 1 6 83
0: 3
1: 5
2: 23
3: -1
4: 456
5: 1
6: 6
7: 83
Max even #: 456
----
Enter numbers:-5 -10 -44 -2 -27 -9 -27 -9
0:-5
1:-10
2:-44
3:-2
4:-27
5:-9
6:-27
7:-9
Max even #: -2
```

IndexError and enumerate()

A common error is to try to access a list with an index that is out of the list's index range, e.g., to try to access my_list[8] when my_list's valid indices are 0-7. Accessing an index that is out of range causes the program to automatically abort execution and generate an ***IndexError***. Ex: For a list my_list containing 8 elements, the statement `my_list[10] = 42` produces output similar to:

```
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

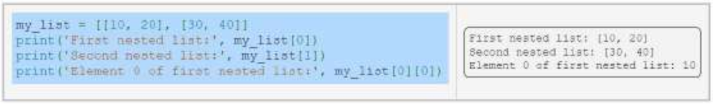
The built-in enumerate() function iterates over a list and provides an iteration counter

Table 11.3.1: Built-in functions supporting list objects.

Function	Description	Example code	Example output
all(list)	True if every element in list is True (!= 0), or if the list is empty.	`print(all([1, 2, 3]))` `print(all([0, 1, 2]))`	True False
any(list)	True if any element in the list is True.	`print(any([0, 2]))` `print(any([0, 0]))`	True False
max(list)	Get the maximum element in the list.	`print(max([-3, 5, 25]))`	25
min(list)	Get the minimum element in the list.	`print(min([-3, 5, 25]))`	-3
sum(list)	Get the sum of all elements in the list.	`print(sum([-3, 5, 25]))`	27

List Nesting

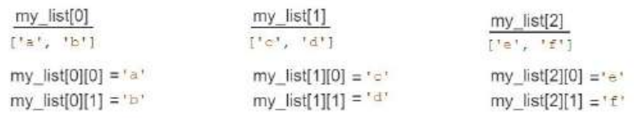
An embedding of a list inside another list is known as **list nesting**



```
my_list = [[10, 20], [30, 40]]
print('First nested list:', my_list[0])
print('Second nested list:', my_list[1])
print('Element 0 of first nested list:', my_list[0][0])
```

```
First nested list: [10, 20]
Second nested list: [30, 40]
Element 0 of first nested list: 10
```

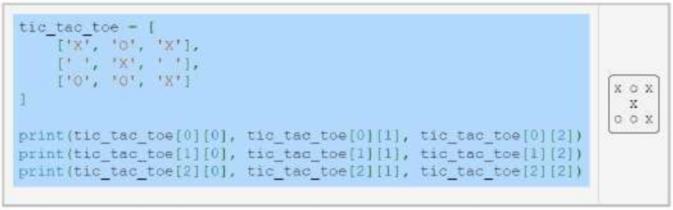
To access the elements of a nested list: my_list[0][0]



```
my_list[0]            my_list[1]            my_list[2]
['a', 'b']            ['c', 'd']            ['e', 'f']

my_list[0][0] = 'a'   my_list[1][0] = 'c'   my_list[2][0] = 'e'
my_list[0][1] = 'b'   my_list[1][1] = 'd'   my_list[2][1] = 'f'
```

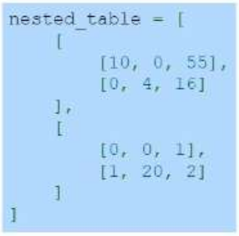
The elements of each nested list can be accessed using two indexing operations.

List nesting allows for a programmer to also create a **multi-dimensional data structure**, the simplest being a two-dimensional table, like a spreadsheet



```
tic_tac_toe = [
    ['X', 'O', 'X'],
    [' ', 'X', ' '],
    ['O', 'O', 'X']
]

print(tic_tac_toe[0][0], tic_tac_toe[0][1], tic_tac_toe[0][2])
print(tic_tac_toe[1][0], tic_tac_toe[1][1], tic_tac_toe[1][2])
print(tic_tac_toe[2][0], tic_tac_toe[2][1], tic_tac_toe[2][2])
```

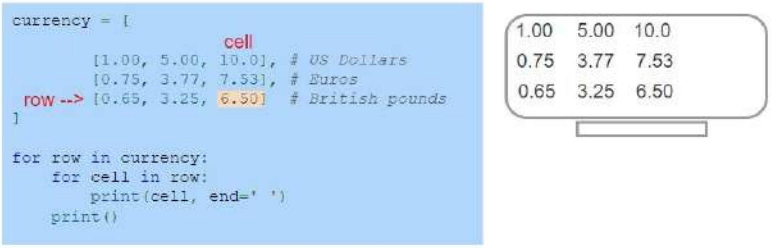
```
X O X
  X
O O X
```



```
nested_table = [
    [
        [10, 0, 55],
        [0, 4, 16]
    ],
    [
        [0, 0, 1],
        [1, 20, 2]
    ]
]
```

The level of nested lists is arbitrary. A programmer might create a three-dimensional list

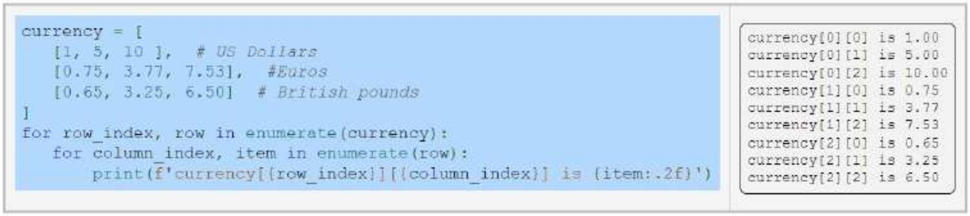
A programmer can access all of the elements of nested lists by using **nested for loops**



```
currency = [
                             cell
        [1.00, 5.00, 10.0],  # US Dollars
        [0.75, 3.77, 7.53],  # Euros
 row --> [0.65, 3.25, 6.50]  # British pounds
]

for row in currency:
    for cell in row:
        print(cell, end=' ')
    print()
```

```
1.00   5.00   10.0
0.75   3.77   7.53
0.65   3.25   6.50
```

Figure 11.5.4: Iterating through multi-dimensional lists using enumerate().

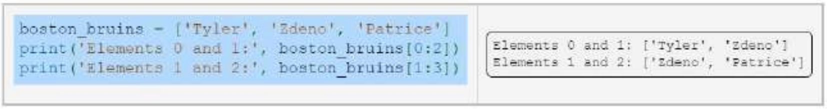


```
currency = [
    [1, 5, 10 ],    # US Dollars
    [0.75, 3.77, 7.53],   #Euros
    [0.65, 3.25, 6.50]   # British pounds
]
for row_index, row in enumerate(currency):
    for column_index, item in enumerate(row):
        print(f'currency[{row_index}][{column_index}] is {item:.2f}')
```

```
currency[0][0] is 1.00
currency[0][1] is 5.00
currency[0][2] is 10.00
currency[1][0] is 0.75
currency[1][1] is 3.77
currency[1][2] is 7.53
currency[2][0] is 0.65
currency[2][1] is 3.25
currency[2][2] is 6.50
```

List Slicing

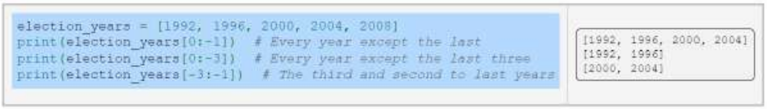
A programmer can use slice notation to read multiple elements from a list, creating a new list that contains only the desired elements



```
boston_bruins = ['Tyler', 'Zdeno', 'Patrice']
print('Elements 0 and 1:', boston_bruins[0:2])
print('Elements 1 and 2:', boston_bruins[1:3])
```

```
Elements 0 and 1: ['Tyler', 'Zdeno']
Elements 1 and 2: ['Zdeno', 'Patrice']
```

Figure 11.6.2: List slicing: Using negative indices.



```
election_years = [1992, 1996, 2000, 2004, 2008]
print(election_years[0:-1])  # Every year except the last
print(election_years[0:-3])  # Every year except the last three
print(election_years[-3:-1])  # The third and second to last years
```

```
[1992, 1996, 2000, 2004]
[1992, 1996]
[2000, 2004]
```

An optional component of slice notation is the stride, which indicates how many elements are skipped between extracted items in the source list

my_list[0:5:2] has a stride of 2, thus skipping every other element

Table 11.6.1: Some common list slicing operations.

Operation	Description	Example code	Example output

my_list[start:end]	Get a list from start to end (minus 1).	`my_list = [5, 10, 20]`
		`print(my_list[0:2])`
			`[5, 10]`

my_list[start:end:stride]	Get a list of every stride element from start to end (minus 1).	`my_list = [5, 10, 20, 40, 80]`
		`print(my_list[0:5:3])`
			`[5, 40]`

my_list[start:]	Get a list from start to end of the list.	`my_list = [5, 10, 20, 40, 80]`
		`print(my_list[2:])`
			`[20, 40, 80]`

my_list[:end]	Get a list from beginning of list to end (minus 1).	`my_list = [5, 10, 20, 40, 80]`
		`print(my_list[:4])`
			`[5, 10, 20, 40]`

my_list[:]	Get a copy of the list.	`my_list = [5, 10, 20, 40, 80]`
		`print(my_list[:])`
			`[5, 10, 20, 40, 80]`

Loops Modifying Lists

Changing elements' values

The below example of changing element's values combines the len() and range() functions to iterate over a list and increment each element of the list by 5.

Figure 11.7.1: Modifying a list during iteration example.

```
my_list = [3.2, 5.0, 16.5, 12.25]

for i in range(len(my_list)):
    my_list[ i ] += 5
```

Figure 11.7.2: Modifying a list during iteration example: Converting negative values to 0.

Correct way to modify the list.

```
user_input = input('Enter numbers: ')

tokens = user_input.split()

# Convert strings to integers
nums = []
for token in tokens:
    nums.append(int(token))

# Print each position and number
print()
for pos, val in enumerate(nums):

    print(f'{pos}: {val}')

# Change negative values to 0
for pos in range(len(nums)):
    if nums[pos] < 0:
        nums[pos] = 0

# Print new numbers
print('New numbers: ')
for num in nums:
    print(num, end=' ')
```

```
Enter numbers:5 67 -5 -4 5 6 6 4
0: 5
1: 67
2: -5
3: -4
4: 5
5: 6
6: 6
7: 4
New numbers:
5 67 0 0 5 6 6 4
```

Incorrect way: list not modified.

```
user_input = input('Enter numbers:')

tokens = user_input.split()

# Convert strings to integers
nums = []
for token in tokens:
    nums.append(int(token))

# Print each position and number
print()
for pos, val in enumerate(nums):

    print(f'{pos}: {val}')

# Change negative values to 0
for num in nums:
    if num < 0:
        num = 0  # Logic error: temp variable num set to 0

# Print new numbers
print('New numbers: ')
for num in nums:
    print(num, end=' ')
```

```
Enter numbers:5 67 -5 -4 5 6 6 4
0: 5
1: 67
2: -5
3: -4
4: 5
5: 6
6: 6
7: 4
New numbers:
5 67 -5 -4 5 6 6 4
```

The program on the right illustrates a common logic error. *A common error when modifying a list during iteration is to update the loop variable instead of the list object.* The statement `num = 0` simply binds the name num to the integer literal value 0. The reference in the list is never changed.

In contrast, the program on the left correctly uses an index operation `nums[pos] = 0` to modify to 0 the reference held by the list in position pos. The below activities demonstrate further; note that only the second program changes the list's values.

```
user_values = [-1, 4, -6, 7]

for n in range(len(user_values)):
  if user_values[n] < 0:
    user_values[n] = -1 * user_values[n]
  print(user_values[n])
```

```
user_values = [2, 5, 8]
sum_value = 0

for i in range(len(user_values)):
  sum_value += user_values[i]

print(sum_value)
```

```
user_values = [2, 5, 7, 3]

max_value = user_values[0]
for n in range(len(user_values)):
  if user_values[n] >= max_value:
    max_value = user_values[n]
    print(max_value)
```

Changing list size

A common error is to add or remove a list element while iterating over that list. Such list modification can lead to unexpected behavior if the programmer is not careful. Ex: Consider the following program that reads in two sets of numbers and attempts to find numbers in the first set that are not in the second set.

Figure 11.7.3: Modifying lists while iterating: Incorrect program.

```
nums1 = []
nums2 = []

user_input = input('Enter first set of numbers: ')
tokens = user_input.split()  # Split into separate strings

# Convert strings to integers
print()
for pos, val in enumerate(tokens):
    nums1.append(int(val))

    print(f'{pos}: {val}')

user_input = input('Enter second set of numbers:')
tokens = user_input.split()

# Convert strings to integers
print()
for pos, val in enumerate(tokens):
    nums2.append(int(val))

    print(f'{pos}: {val}')

# Remove elements from nums1 if also in nums2
print()
for val in nums1:
    if val in nums2:

        print(f'Deleting {val}')
        nums1.remove(val)

# Print new numbers
print('\nNumbers only in first set:', end=' ')
for num in nums1:
    print(num, end=' ')
```

```
Enter first set of numbers:5 10 15 20
0: 5
1: 10
2: 15
3: 20
Enter second set of numbers:15 20 25 30
0: 15
1: 20
2: 25
3: 30

Deleting 15

Numbers only in first set: 5 10 20
```

The above example iterates over the list nums1, deleting an element from the list if the element is also found in the list nums2. The programmer expected a certain result, namely that after removing an element from the list, the next iteration of the loop would reference the next element as normal. However, removing the element shifts the position of each following element in the list to the left by one. In the example above, removing 15 from nums1 shifts the value 20 left into position 2. The loop, having just iterated over position 2 and removing 15, moves to the next position and finds the end of the list, thus never evaluating the final value 20.

Figure 11.7.4: Copy a list using [:].

```
for item in my_list[:]:
    # Loop statements.
```

List Comprehensions

A programmer commonly wants to modify every element of a list in the same way, such as adding 10 to every element. A list comprehension iterates over a list, modifying each element, and returns a new list consisting of the modified elements

```
new_list = [expression for loop_variable_name in iterable]
```

A list comprehension has three components:

1. An *expression component* to evaluate for each element in the iterable object.
2. A *loop variable component* to bind to the current iteration element.
3. An *iterable object component* to iterate over (list, string, tuple, enumerate, etc).

A list comprehension is always surrounded by brackets, which is a helpful reminder that the comprehension builds and returns a new list object. The loop variable and iterable object components make up a normal for loop expression. The for loop iterates through the iterable object as normal, and the expression operates on the loop variable in each iteration. The result is a new list containing the values modified by the expression. The below program demonstrates a simple list comprehension that increments each value in a list by 5.

Figure 11.8.1: List comprehension example: A first look.

```
my_list = [10, 20, 30]
list_plus_5 = [(i + 5) for i in my_list]

print('New list contains:', list_plus_5)
```

```
New list contains: [15, 25, 35]
```

Table 11.8.1: List comprehensions can replace some for loops.

Num	Description	For loop	Equivalent list comprehension	Output of both programs

1	Add 10 to every element.	`my_list = [5, 20, 50]`
		`for i in range(len(my_list)):`
		`    my_list[ i ] += 10`
		`print(my_list)`
			`my_list = [5, 20, 50]`
			`my_list = [(i+10) for i in my_list]`
			`print(my_list)`
				`[15, 30, 60]`

2	Convert every element to a string.	`my_list = [5, 20, 50]`
		`for i in range(len(my_list)):`
		`    my_list[ i ] = str(my_list[ i ])`
		`print(my_list)`
			`my_list = [5, 20, 50]`
			`my_list = [str(i) for i in my_list]`
			`print(my_list)`
				`['5', '20', '50']`

3	Convert user input into a list of integers.	`inp = input('Enter numbers:')`
		`my_list = []`
		`for i in inp.split():`
		`    my_list.append(int(i))`
		`print(my_list)`
			`inp = input('Enter numbers:')`
			`my_list = [int(i) for i in inp.split()]`
			`print(my_list)`
				`Enter numbers: 7 9 3`
				`[7, 9, 3]`

4	Find the sum of each row in a two-dimensional list.	`my_list = [[5, 10, 15], [2, 3, 16], [100]]`
		`sum_list = []`
		`for row in my_list:`
		`    sum_list.append(sum(row))`
		`print(sum_list)`
			`my_list = [[5, 10, 15], [2, 3, 16], [100]]`
			`sum_list = [sum(row) for row in my_list]`
			`print(sum_list)`
				`[30, 21, 100]`

5	Find the sum of the row with the smallest sum in a two-dimensional table.	`my_list = [[5, 10, 15], [2, 3, 16], [100]]`
		`sum_list = []`
		`for row in my_list:`
		`    sum_list.append(sum(row))`
		`min_row = min(sum_list)`
		`print(min_row)`
			`my_list = [[5, 10, 15], [2, 3, 16], [100]]`
			`min_row = min([sum(row) for row in my_list])`
			`print(min_row)`
				`21`

Conditional list comprehensions

A list comprehension can be extended with an optional conditional clause that causes the statement to return a list with only certain elements.

Construct 11.8.2: Conditional list comprehensions.

```
new_list = [expression for name in iterable if condition]
```

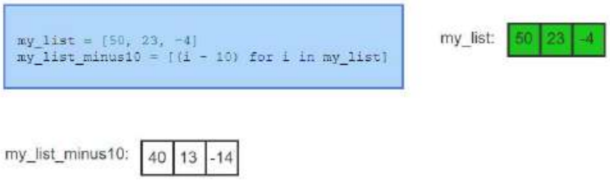
Need help? Feedback?

Using the above syntax will only add an element to the resulting list if the condition evaluates to True. The following program demonstrates list comprehension with a conditional clause that returns a list with only even numbers.

Figure 11.8.2: Conditional list comprehension example: Return a list of even numbers.

```
# Get a list of integers from the user
numbers = [int(i) for i in input('Enter numbers:').split()]

# Return a list of only even numbers
even_numbers = [i for i in numbers if (i % 2) == 0]
print('Even numbers only:', even_numbers)
```

```
Enter numbers: 5 52 16 7 25
Even numbers only: [52, 16]
....
Enter numbers: 8 12 -14 9 0
Even numbers only: [8, 12, -14, 0]
```



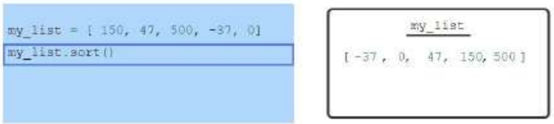
```
my_list = [50, 23, -4]
my_list_minus10 = [(i - 10) for i in my_list]
```

```
my_list = [-3, -2, -1, 0, 1, 2, 3]
new_list = [ number * 3 for number in my_list if (number <= 1) and (number % 2 == 1) ]
print(new_list)
```

```
my_list = [-3, -2, -1, 0, 1, 2, 3]
new_list = [ number * 3 for number in my_list if (number <= 1) and (number % 2 == 1) ]
print(new_list)
```

Sorting Lists

One of the most useful list methods is sort(), sorting the elements from lowest to highest



```
my_list = [ 150, 47, 500, -37, 0]
my_list.sort()
```

```
my_list
[ -37 ,  0,  47,  150,  500 ]
```

x.sort(reverse = True)

Sorting also supports the reverse argument

The sort() method performs element-by-element comparison to determine the final ordering. Numeric type elements like int and float have their values directly compared to determine relative ordering, i.e., 5 is less than 10.

The below program illustrates the basic usage of the list.sort() method, reading book titles into a list and sorting the list alphabetically.

Figure 11.9.1: list.sort() method example: Alphabetically sorting book titles.

```python
books = []
prompt = 'Enter new book: '
user_input = input(prompt).strip()

while (user_input.lower() != 'exit'):
    books.append(user_input)
    user_input = input(prompt).strip()

books.sort()

print('\nAlphabetical order:')
for book in books:
    print(book)
```

```
Enter new book: Pride, Prejudice, and Zombies
Enter new book: Programming in Python
Enter new book: Hackers and Painters
Enter new book: World War Z
Enter new book: exit

Alphabetical order:
Hackers and Painters
Pride, Prejudice, and Zombies
Programming in Python
World War Z
```

The sorted() built-in function provides the same sorting functionality as the list.sort() method, however it creates and returns a new list instead of modifying an existing list

Figure 11.9.2: Using sorted() to create a new sorted list from an existing list without modifying the existing list.

```python
numbers = [int(i) for i in input('Enter numbers: ').split()]

sorted_numbers = sorted(numbers)

print('\nOriginal numbers:', numbers)
print('Sorted numbers:', sorted_numbers)
```

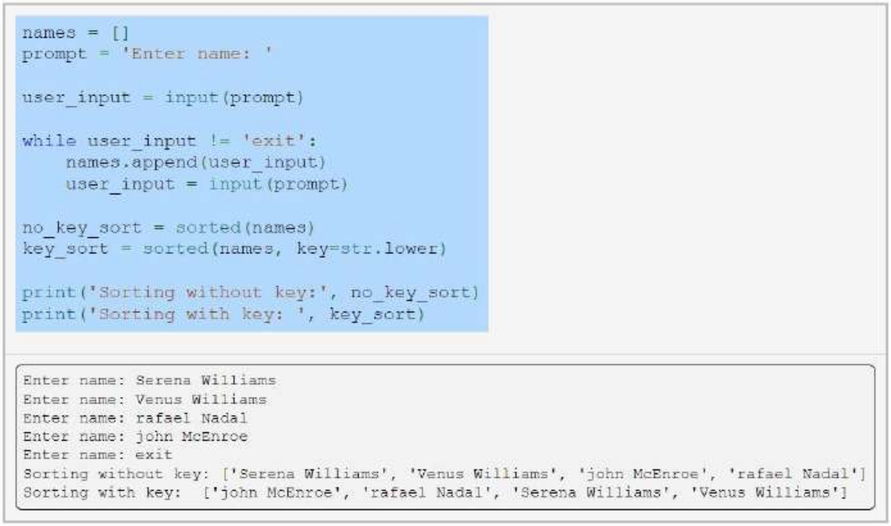
```
Enter numbers: -5 5 -100 23 4 5
Original numbers: [-5, 5, -100, 23, 4, 5]
Sorted numbers: [-100, -5, 4, 5, 5, 23]
```

Both the list.sort() method and the built-in sorted() function have an optional key argument. The key specifies a function to be applied to each element prior to being compared

Figure 11.9.4: The key argument to list.sort() or sorted() can be assigned any function.

```python
my_list = [[25], [15, 25, 35], [10, 15]]

sorted_list = sorted(my_list, key=max)

print('Sorted list:', sorted_list)
```

```
Sorted list: [[10, 15], [25], [15, 25, 35]]
```



```
names = []
prompt = 'Enter name: '

user_input = input(prompt)

while user_input != 'exit':
    names.append(user_input)
    user_input = input(prompt)

no_key_sort = sorted(names)
key_sort = sorted(names, key=str.lower)

print('Sorting without key:', no_key_sort)
print('Sorting with key: ', key_sort)
```

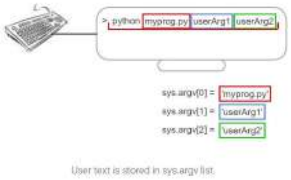
```
Enter name: Serena Williams
Enter name: Venus Williams
Enter name: rafael Nadal
Enter name: john McEnroe
Enter name: exit
Sorting without key: ['Serena Williams', 'Venus Williams', 'john McEnroe', 'rafael Nadal']
Sorting with key:  ['john McEnroe', 'rafael Nadal', 'Serena Williams', 'Venus Williams']
```

Command-line Arguments

Command-line arguments are values entered by a user when running a program from a command line

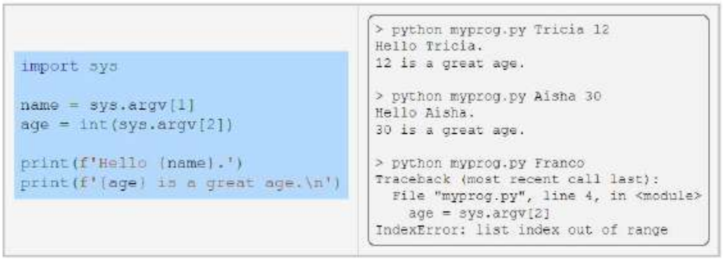
The contents of this command line are automatically stored in the list sys.argv, which is stored in the standard library sys module.

Sys.argv consists of one string element for each argument typed on the command line



> python myprog.py userArg1 userArg2
sys.argv[0] = 'myprog.py'
sys.argv[1] = 'userArg1'
sys.argv[2] = 'userArg2'
User text is stored in sys.argv list.

Figure 11.10.1: Simple use of command line arguments.



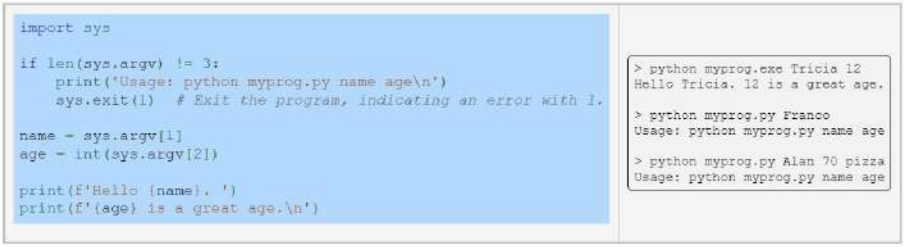
```
import sys

name = sys.argv[1]
age = int(sys.argv[2])

print(f'Hello {name}.')
print(f'{age} is a great age.\n')
```

```
> python myprog.py Tricia 12
Hello Tricia.
12 is a great age.

> python myprog.py Aisha 30
Hello Aisha.
30 is a great age.

> python myprog.py Franco
Traceback (most recent call last):
  File "myprog.py", line 4, in <module>
    age = sys.argv[2]
IndexError: list index out of range
```

While a program may expect the user to enter certain command-line arguments, there is no guarantee that the user will do so. *A common error is to access elements within argv without first checking the length of argv to ensure that the user entered enough arguments, resulting in an IndexError being generated.* In the last example above, the user did not enter the age argument, resulting in an IndexError when accessing argv. Conversely, if a user entered too many arguments, extra arguments will be ignored. Above, if the user typed `python myprog.py Alan 70 pizza`, "pizza" will be stored in argv[3] but will never be used by the program.

Thus, when a program uses command-line arguments, a good practice is to always check the length of argv at the beginning of the program to ensure that the user entered the correct number of arguments. The following program uses the statement `if len(sys.argv) != 3` to check for the correct number of arguments, the three arguments being the program, name, and age. If the number of arguments is incorrect, the program prints an error message, referred to as a ***usage message***, that provides the user with an example of the correct command-line argument format. *A good practice is to always output a usage message when the user enters incorrect command-line arguments.*

Figure 11.10.2: Checking for proper number of command-line arguments.



```
import sys

if len(sys.argv) != 3:
    print('Usage: python myprog.py name age\n')
    sys.exit(1)  # Exit the program, indicating an error with 1.

name = sys.argv[1]
age = int(sys.argv[2])

print(f'Hello {name}. ')
print(f'{age} is a great age.\n')
```

```
> python myprog.exe Tricia 12
Hello Tricia. 12 is a great age.

> python myprog.py Franco
Usage: python myprog.py name age

> python myprog.py Alan 70 pizza
Usage: python myprog.py name age
```

Dictionaries

The dict type implements a dictionary in Python

There are several approaches to create a dict:

- The first approach wraps braces { } around key-value pairs of literals and/or variables: `{'Jose': 'A+', 'Gino': 'C-'}` creates a dictionary with two keys 'Jose' and 'Gino' that are associated with the grades 'A+' and 'C-', respectively.
- The second approach uses ***dictionary comprehension***, which evaluates a loop to create a new dictionary, similar to how list comprehension creates a new list. Dictionary comprehension is out of scope for this material.
- Other approaches use the ***dict()*** built-in function, using either keyword arguments to specify the key-value pairs or by specifying a list of tuple-pairs. The following creates equivalent dictionaries:
  - dict(Bobby='805-555-2232', Johnny='951-555-0055')
  - dict([('Bobby', '805-555-2232'), ('Johnny', '951-555-0055')])

Operation	Description	Example code
my_dict[key]	Indexing operation – retrieves the value associated with key.	`jose_grade = my_dict['Jose']`
my_dict[key] = value	Adds an entry if the entry does not exist, else modifies the existing entry.	`my_dict['Jose'] = 'B+'`
del my_dict[key]	Deletes the key from a dict.	`del my_dict['Jose']`
key in my_dict	Tests for existence of key in my_dict.	`if 'Jose' in my_dict: # ....`

```
provincial_capitals = {
    'Yukon': 'Whitehorse',
    'BC': 'Victoria',
    'Alberta': 'Edmonton',
    'Nunavut': 'Iqaluit'
}

province_name = input()
while province_name != 'exit':
    if province_name in provincial_capitals:
        print(provincial_capitals[province_name])
    else:
        print('x')
    province_name = input()
```

Dictionary Methods

A **dictionary method** is a function provided by the dictionary type (dict) that operates on a specific dictionary object

Dictionary method	Description	Code example	Output

my_dict.clear()	Removes all items from the dictionary.	`my_dict = {'Ahmad': 1, 'Jane': 42}`
		`my_dict.clear()`
		`print(my_dict)`
			`{}`

my_dict.get(key, default)	Reads the value of the key from the dictionary. If the key does not exist in the dictionary, then returns default.	`my_dict = {'Ahmad': 1, 'Jane': 42}`
		`print(my_dict.get('Jane', 'N/A'))`
		`print(my_dict.get('Chad', 'N/A'))`
			`42`
			`N/A`

my_dict1.update(my_dict2)	Merges dictionary my_dict1 with another dictionary my_dict2. Existing entries in my_dict1 are overwritten if the same keys exist in my_dict2.	`my_dict = {'Ahmad': 1, 'Jane': 42}`
		`my_dict.update({'John': 50})`
		`print(my_dict)`
			`{'Ahmad': 1,`
			`'Jane': 42,`
			`'John': 50}`

my_dict.pop(key, default)	Removes and returns the key value from the dictionary. If key does not exist, then default is returned.	`my_dict = {'Ahmad': 1, 'Jane': 42}`
		`val = my_dict.pop('Ahmad')`
		`print(my_dict)`
			`{'Jane': 42}`

```
airport_codes = {
    'Houston': 'IAH',
    'Minneapolis': 'MSP',
    'Washington': 'IAD',
    'Amsterdam': 'AMS',
    'New York': 'JFK'
}

new_airport_codes = {
    'Houston': 'HOU',
    'Los Angeles': 'LAX',
    'Seattle': 'SEA'
}

print(airport_codes.get('Houston', 'na'))
print(airport_codes.get('Los Angeles', 'na'))
airport_codes.update(new_airport_codes)
print(airport_codes.get('Houston', 'na'))
print(airport_codes.get('Los Angeles', 'na'))
```

```
IAH
na
HOU
LAX
```

```
airport_codes = {
    'Dallas': 'DAL',
    'Cincinnati': 'CVG',
    'Los Angeles': 'LAX',
    'Chicago': 'ORD',
    'Amsterdam': 'AMS'
}

new_airport_codes = {
    'Minneapolis': 'MSP',
    'San Francisco': 'SFO',
    'Austin': 'AUS'
}

print(airport_codes.get('Seattle', 'na'))
print(airport_codes.get('Cincinnati', 'na'))
print(airport_codes.get('Austin', 'na'))
airport_codes.update(new_airport_codes)
print(airport_codes.get('Seattle', 'na'))
print(airport_codes.get('Cincinnati', 'na'))
print(airport_codes.get('Austin', 'na'))
```

```
na
CVG
na
na
CVG
AUS
```

Iterating Over a Dictionary

A hash is a transformation of the key into a unique value that allows the interpreter to perform very fast lookup

```
for key in dictionary:  # Loop expression
    # Statements to execute in the loop

#Statements to execute after the loop
```

The dict type also supports the useful methods items(), keys(), and values() methods, which produce a view object. A ***view object*** provides read-only access to dictionary keys and values. A program can iterate over a view object to access one key-value pair, one key, or one value at a time, depending on the method used. A view object reflects any updates made to a dictionary, even if the dictionary is altered after the view object is created.

- dict.items() – returns a view object that yields (key, value) tuples.
- dict.keys() – returns a view object that yields dictionary keys.
- dict.values() – returns a view object that yields dictionary values.

dict.items()

```
num_calories = dict(Coke=90, Coke_zero=0,
Pepsi=94)
for soda, calories in num_calories.items():
    print(f'{soda}: {calories}')
```

```
Coke: 90
Coke_zero: 0
Pepsi: 94
```

dict.keys()

```
num_calories = dict(Coke=90, Coke_zero=0,
Pepsi=94)
for soda in num_calories.keys():
    print(soda)
```

```
Coke
Coke_zero
Pepsi
```

dict.values()

```
num_calories = dict(Coke=90, Coke_zero=0,
Pepsi=94)
for soda in num_calories.values():
    print(soda)
```

```
90
0
94
```

When a program iterates over a view object, one result is generated for each iteration as needed, instead of generating an entire list containing all of the keys or values. Such behavior allows the interpreter to save memory. Since results are generated as needed, view objects do not support indexing. A statement such as my_dict.keys()[0] produces an error. Instead, a valid approach is to use the list() built-in function to convert a view object into a list, and then perform the necessary operations. The example below converts a dictionary view into a list, so that the list can be sorted to find the first two closest planets to Earth.

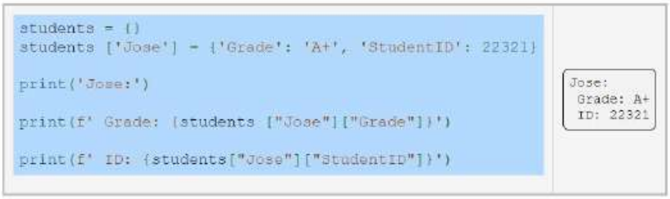
Figure 11.14.2: Use list() to convert view objects into lists.

```
solar_distances = dict(mars=219.7e6, venus=116.4e6, jupiter=546e6,
pluto=2.95e9)
list_of_distances = list(solar_distances.values())  # Convert view to list

sorted_distance_list = sorted(list_of_distances)
closest = sorted_distance_list[0]
next_closest = sorted_distance_list[1]

print(f'Closest planet is {closest:.4e}')
print(f'Second closest planet is {next_closest:.4e}')
```

```
Closest planet is 1.1640e+08
Second closest planet is
2.1970e+08
```

Dictionary Nesting

A dictionary may contain one or more nested dictionaries, in which the dictionary contains another dictionary as a value

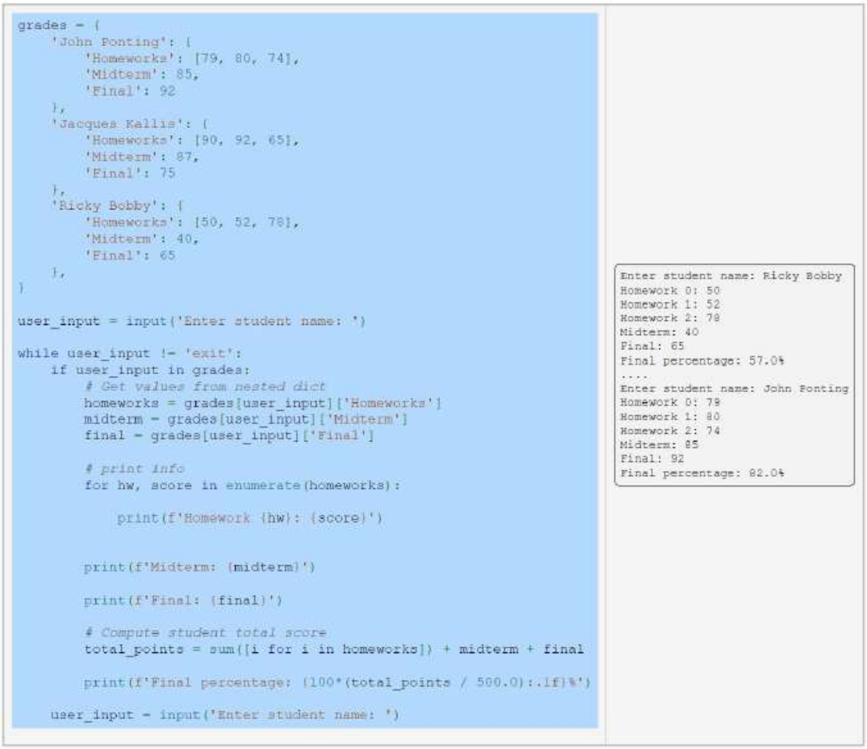


```
students = {}
students ['Jose'] = {'Grade': 'A+', 'StudentID': 22321}

print('Jose:')

print(f' Grade: {students ["Jose"]["Grade"]}')

print(f' ID: {students["Jose"]["StudentID"]}')
```

```
Jose:
  Grade: A+
  ID: 22321
```

The variable students is first created as an empty dictionary. An indexing operation creates a new entry in students with the key 'Jose' and the value of another dictionary. Indexing operations can be applied to the nested dictionary by using consecutive sets of brackets []. The expression `students['Jose']['Grade']` first obtains the value of the key 'Jose' from students, yielding the nested dictionary. The second set of brackets indexes into the nested dictionary, retrieving the value of the key 'Grade'.

Nested dictionaries also serve as a simple but powerful data structure. Data structure is a method of organizing data in a logical and coherent fashion



```
grades = {
    'John Ponting': {
        'Homeworks': [79, 80, 74],
        'Midterm': 85,
        'Final': 92
    },
    'Jacques Kallis': {
        'Homeworks': [90, 92, 65],
        'Midterm': 87,
        'Final': 75
    },
    'Ricky Bobby': {
        'Homeworks': [50, 52, 78],
        'Midterm': 40,
        'Final': 65
    },
}

user_input = input('Enter student name: ')

while user_input != 'exit':
    if user_input in grades:
        # Get values from nested dict
        homeworks = grades[user_input]['Homeworks']
        midterm = grades[user_input]['Midterm']
        final = grades[user_input]['Final']

        # print info
        for hw, score in enumerate(homeworks):

            print(f'Homework {hw}: {score}')


        print(f'Midterm: {midterm}')

        print(f'Final: {final}')

        # Compute student total score
        total_points = sum([i for i in homeworks]) + midterm + final

        print(f'Final percentage: {100*(total_points / 500.0):.1f}%')

    user_input = input('Enter student name: ')
```

```
Enter student name: Ricky Bobby
Homework 0: 50
Homework 1: 52
Homework 2: 78
Midterm: 40
Final: 65
Final percentage: 57.0%
....
Enter student name: John Ponting
Homework 0: 79
Homework 1: 80
Homework 2: 74
Midterm: 85
Final: 92
Final percentage: 82.0%
```

Note the whitespace and indentation used to layout the nested dictionaries. Such layout improves the readability of the code and makes the hierarchy of the data structure obvious. The extra whitespace does not affect the dict elements, as the interpreter ignores indentation in a multi-line construct.

A benefit of using nested dictionaries is that the code tends to be more readable, especially if the keys are a category like 'Homeworks'. Alternatives like nested lists tend to require more code, consisting of more loops constructs and variables.

Dictionaries support arbitrary levels of nesting; Ex: The expression `students['Jose']['Homeworks'][2]['Grade']` might be applied to a dictionary that has four levels of nesting.

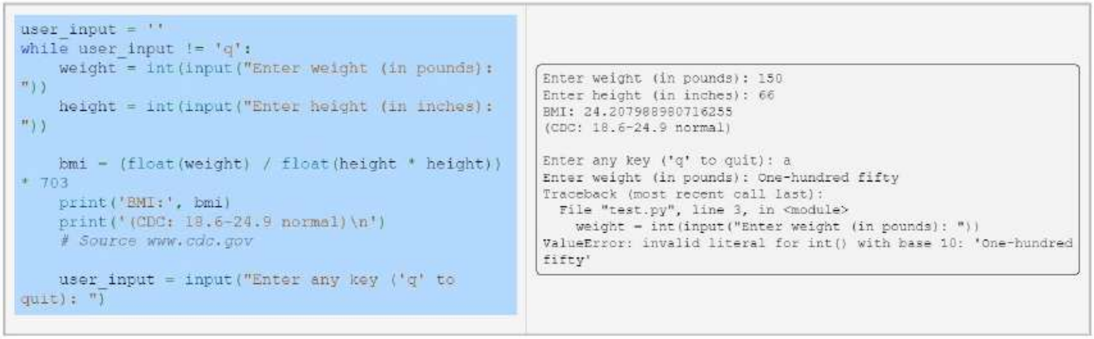
Exceptions

Handling Exceptions using Try and Except

Error-checking code is code that a programmer introduces to detect and handle errors that may occur while the program executes

Python has special constructs known as exception-handling constructs because they handle exceptional circumstances (errors during execution)

Figure 12.1.1: BMI example without exception handling.

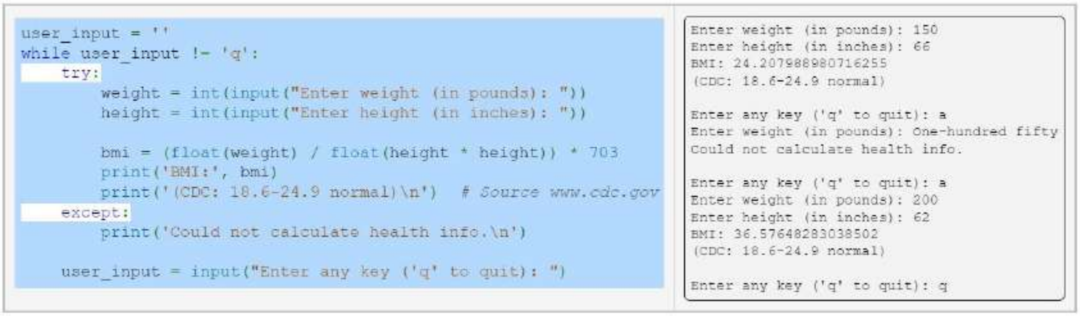


```
user_input = ''
while user_input != 'q':
    weight = int(input("Enter weight (in pounds): "))
    height = int(input("Enter height (in inches): "))

    bmi = (float(weight) / float(height * height)) * 703
    print('BMI:', bmi)
    print('(CDC: 18.6-24.9 normal)\n')
    # Source www.cdc.gov

    user_input = input("Enter any key ('q' to quit): ")
```

```
Enter weight (in pounds): 150
Enter height (in inches): 66
BMI: 24.207988980716255
(CDC: 18.6-24.9 normal)

Enter any key ('q' to quit): a
Enter weight (in pounds): One-hundred fifty
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    weight = int(input("Enter weight (in pounds): "))
ValueError: invalid literal for int() with base 10: 'One-hundred fifty'
```

Code that potentially may produce an exception is placed in a **try** block. If the code in the try block causes an exception, then the code placed in a following **except** block is executed

Figure 12.1.2: BMI example with exception handling using try/except.



```
user_input = ''
while user_input != 'q':
    try:
        weight = int(input("Enter weight (in pounds): "))
        height = int(input("Enter height (in inches): "))

        bmi = (float(weight) / float(height * height)) * 703
        print('BMI:', bmi)
        print('(CDC: 18.6-24.9 normal)\n')   # Source www.cdc.gov
    except:
        print('Could not calculate health info.\n')

    user_input = input("Enter any key ('q' to quit): ")
```

```
Enter weight (in pounds): 150
Enter height (in inches): 66
BMI: 24.207988980716255
(CDC: 18.6-24.9 normal)

Enter any key ('q' to quit): a
Enter weight (in pounds): One-hundred fifty
Could not calculate health info.

Enter any key ('q' to quit): a
Enter weight (in pounds): 200
Enter height (in inches): 62
BMI: 36.57648283038502
(CDC: 18.6-24.9 normal)

Enter any key ('q' to quit): q
```

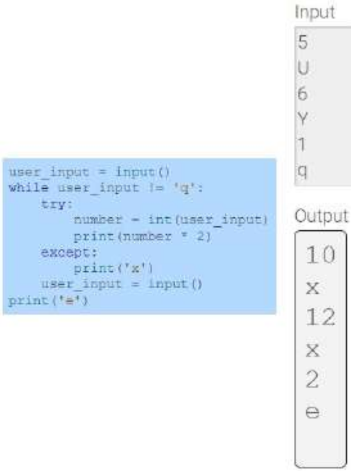
The try and except constructs are used together to implement **exception handling**, meaning handling exceptional conditions (errors during execution)

Construct 12.1.1: Basic exception handling constructs.

```
try:
    # ... Normal code that might produce errors
except: # Go here if *any* error occurs in try block
    # ... Exception handling code
```

Table 12.1.1: Common exception types.

Type	Reason exception is raised
EOFError	input() hits an end-of-file condition (EOF) without reading any input
KeyError	A dictionary key is not found in the set of keys
ZeroDivisionError	Divide by zero error
ValueError	Invalid value (Ex: Input mismatch)
IndexError	Index out of bounds



```
user_input = input()
while user_input != 'q':
    try:
        number = int(user_input)
        print(number * 2)
    except:
        print('x')
    user_input = input()
print('e')
```

Input

```
5
U
6
Y
1
q
```

Output

```
10
x
12
x
2
e
```

Multiple Exception Handlers

Multiple exception handlers can be added to a try block by adding additional except blocks and specifying the specific type of exception that each except block handles

```
try:
    # ... Normal code
except exceptiontype1:
    # ... Code to handle exceptiontype1
except exceptiontype2:
    # ... Code to handle exceptiontype2
...
except:
    # ... Code to handle other exception types
```

If no exception handler exists for an error type, then an unhandled exception may occur. An unhandled exception causes the interpreter to print the exception that occured and then halt

Figure 12.2.1: BMI example with multiple exception types.

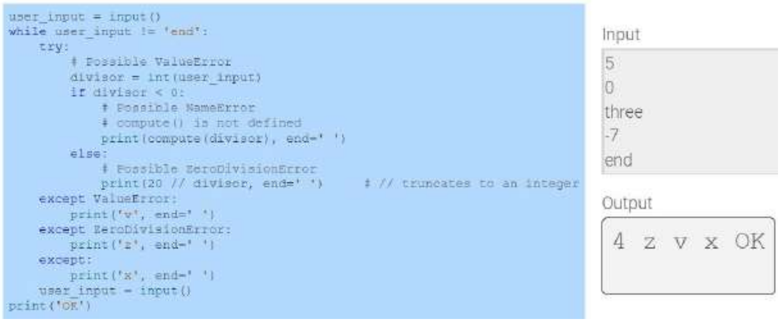
```
user_input = ''
while user_input != 'q':
    try:
        weight = int(input("Enter weight (in pounds): "))
        height = int(input("Enter height (in inches): "))

        bmi = (float(weight) / float(height * height)) * 703
        print('BMI:', bmi)
        print('(CDC: 18.6-24.9 normal)\n')  # Source www.cdc.gov
    except ValueError:
        print('Could not calculate health info.\n')
    except ZeroDivisionError:
        print('Invalid height entered. Must be > 0.')

    user_input = input("Enter any key ('q' to quit): ")
```

```
Enter weight (in pounds): 150
Enter height (in inches): 66
BMI: 24.207988980716255
(CDC: 18.6-24.9 normal)

Enter any key ('q' to quit): a
Enter weight (in pounds): One-hundred fifty
Could not calculate health info.

Enter any key ('q' to quit): a
Enter weight (in pounds): 150
Enter height (in inches): 0
Invalid height entered. Must be > 0.
Enter any key ('q' to quit): q
```

A tuple can be used to specify all of the exception types for which a handler's code should be executed

Figure 12.2.2: Multiple exception types in a single exception handler.

```
try:
    # ...
except (ValueError, TypeError):
    # Exception handler for any ValueError or TypeError that occurs.
except (NameError, AttributeError):
    # A different handler for NameError and AttributeError exceptions.
except:
    # A different handler for any other exception type.
```



```
user_input = input()
while user_input != 'end':
    try:
        # Possible ValueError
        divisor = int(user_input)
        if divisor < 0:
            # Possible NameError
            # compute() is not defined
            print(compute(divisor), end=' ')
        else:
            # Possible ZeroDivisionError
            print(20 // divisor, end=' ')     # // truncates to an integer
    except ValueError:
        print('v', end=' ')
    except ZeroDivisionError:
        print('z', end=' ')
    except:
        print('x', end=' ')
    user_input = input()
print('OK')
```

Input

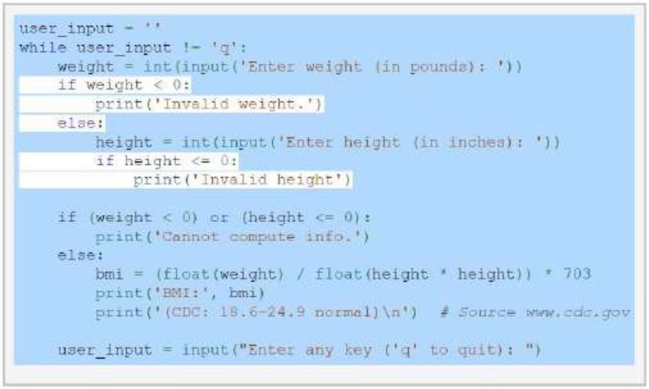
```
5
0
three
-7
end
```

Output

```
4 z v x OK
```

Raising Exceptions

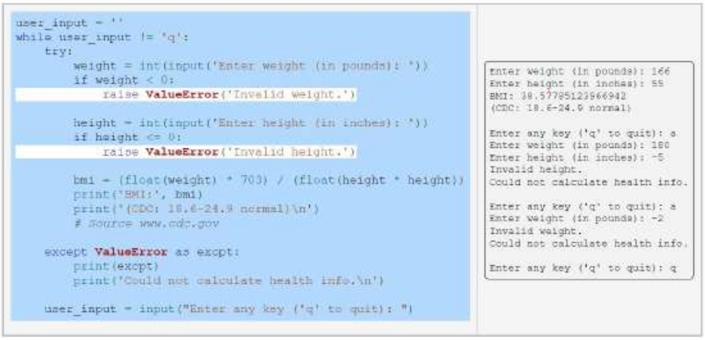
Figure 12.3.1: BMI example with error-checking code but without using exception-handling constructs.



```
user_input = ''
while user_input != 'q':
    weight = int(input('Enter weight (in pounds): '))
    if weight < 0:
        print('Invalid weight.')
    else:
        height = int(input('Enter height (in inches): '))
        if height <= 0:
            print('Invalid height')

    if (weight < 0) or (height <= 0):
        print('Cannot compute info.')
    else:
        bmi = (float(weight) / float(height * height)) * 703
        print('BMI:', bmi)
        print('(CDC: 18.6-24.9 normal)\n')  # Source www.cdc.gov

    user_input = input("Enter any key ('q' to quit): ")
```

The following program shows the same error-checking. Code that executes an error can execute a raise statement, which causes immediate exit from the try block and the execution of an exception handler

Figure 12.3.2: BMI example with error-checking code that raises exceptions.



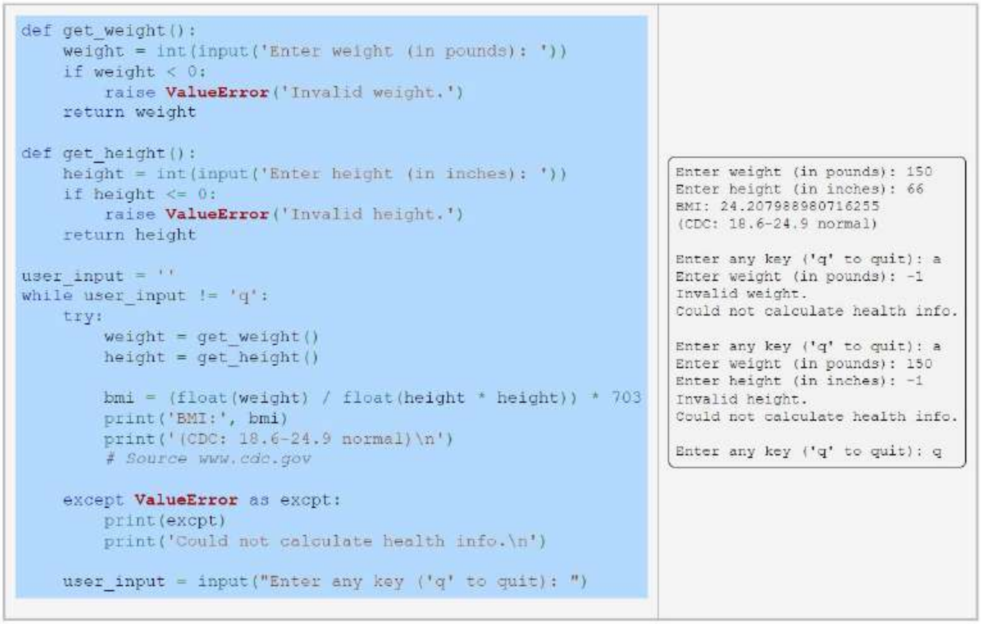
```
user_input = ''
while user_input != 'q':
    try:
        weight = int(input('Enter weight (in pounds): '))
        if weight < 0:
            raise ValueError('Invalid weight.')

        height = int(input('Enter height (in inches): '))
        if height <= 0:
            raise ValueError('Invalid height.')

        bmi = (float(weight) * 703) / (float(height * height))
        print('BMI:', bmi)
        print('(CDC: 18.6-24.9 normal)\n')
        # Source www.cdc.gov

    except ValueError as excpt:
        print(excpt)
        print('Could not calculate health info.\n')

    user_input = input("Enter any key ('q' to quit): ")
```

```
Enter weight (in pounds): 166
Enter height (in inches): 55
BMI: 38.57785123966942
(CDC: 18.6-24.9 normal)

Enter any key ('q' to quit): a
Enter weight (in pounds): 180
Enter height (in inches): -5
Invalid height.
Could not calculate health info.

Enter any key ('q' to quit): a
Enter weight (in pounds): -2
Invalid weight.
Could not calculate health info.

Enter any key ('q' to quit): q
```

The as keyword binds a name to the exception being handled - creates a new variable that might inspect for details about the exception instance

Exceptions with Functions

Figure 12.4.1: BMI example using exception-handling constructs along with functions.



```python
def get_weight():
    weight = int(input('Enter weight (in pounds): '))
    if weight < 0:
        raise ValueError('Invalid weight.')
    return weight

def get_height():
    height = int(input('Enter height (in inches): '))
    if height <= 0:
        raise ValueError('Invalid height.')
    return height

user_input = ''
while user_input != 'q':
    try:
        weight = get_weight()
        height = get_height()

        bmi = (float(weight) / float(height * height)) * 703
        print('BMI:', bmi)
        print('(CDC: 18.6-24.9 normal)\n')
        # Source www.cdc.gov

    except ValueError as excpt:
        print(excpt)
        print('Could not calculate health info.\n')

    user_input = input("Enter any key ('q' to quit): ")
```

```
Enter weight (in pounds): 150
Enter height (in inches): 66
BMI: 24.207988980716255
(CDC: 18.6-24.9 normal)

Enter any key ('q' to quit): a
Enter weight (in pounds): -1
Invalid weight.
Could not calculate health info.

Enter any key ('q' to quit): a
Enter weight (in pounds): 150
Enter height (in inches): -1
Invalid height.
Could not calculate health info.

Enter any key ('q' to quit): q
```

The power of exceptions becomes even more clear when used within functions. If an exception is raised within a function and is not handled within that function, then the function is immediately exited and the calling function is checked for a handler, and so on up the function call hierarchy. The following program illustrates. Note the clarity of the normal code, which obviously "gets the weight, gets the height, and prints the BMI" – the error checking code does not obscure the normal code.

Suppose get_weight() raises an exception of type ValueError. The get_weight() function does not handle exceptions (there is no try block in the function) so it immediately exits. Going up the function call hierarchy returns execution to the global scope script code, where the call to get_weight() was in a try block, so the exception handler for ValueError is executed.

Notice the clarity of the script's code. Without exceptions, the get_weight() function would have had to somehow indicate failure, perhaps through a special return value like -1. The script would have had to check for such failure and would have required additional if-else statements, obscuring the functionality of the code.

Using Finally to Clean Up

The finally clause of a try statement allows a programmer to specify clean-up actions that are always executed

The finally clause is always the last code executed before the try block finishes.

- If *no exception* occurs, then execution continues in the finally clause, and then proceeds with the rest of the program.
- If a *handled exception* occurs, then an exception handler executes and then the finally clause.
- If an *unhandled exception* occurs, then the finally clause executes and then the exception is re-raised.
- The finally clause also executes if any break, continue, or return statement causes the try block to be exited.

The finally clause can be combined with exception handlers, provided that the finally clause comes last. The following program attempts to read integers from a file. The finally clause is always executed, even if some exception occurs when reading the data (such as if the file contains letters, thus causing int() to raise an exception, or if the file does not exist).

Figure 12.5.1: Clean-up actions using finally.

```
nums = []
rd_nums = -1
my_file = input('Enter file name: ')

try:
    print('Opening', my_file)
    rd_nums = open(my_file, 'r')  # Might cause IOError

    for line in rd_nums:
        nums.append(int(line))  # Might cause ValueError
except IOError:
    print('Could not find', my_file)
except ValueError:
    print('Could not read number from', my_file)
finally:
    print('Closing', my_file)
    if rd_nums != -1:
        rd_nums.close()
    print('Numbers found:', ' '.join([str(n) for n in nums]))
```

```
Enter file name: myfile.txt
Opening myfile.txt
Closing myfile.txt
Numbers found: 5 423 234
...
Enter file name: myfile.txt
Opening myfile.txt
Could not read number from myfile.txt
Closing myfile.txt
Numbers found:
...
Enter file name: invalidfile.txt
Opening invalidfile.txt
Could not find invalidfile.txt
Closing invalidfile.txt
Numbers found:
```

Custom Exception Types

When raising an exception, a programmer can use the existing built-in exception types. For example, if an exception should be raised when the value of my_num is less than 0, the programmer might use a ValueError, as in `raise ValueError("my_num < 0")`. Alternatively, a ***custom exception type*** can be defined and then raised. The following example shows how a custom exception type LessThanZeroError might be used.

Figure 12.6.1: Custom exception types.

```
# Define a custom exception type
class LessThanZeroError(Exception):
    def __init__(self, value):
        self.value = value

my_num = int(input('Enter number: '))

if my_num < 0:
    raise LessThanZeroError('my_num must be greater than 0')
else:
    print('my_num:', my_num)
```

```
Enter number: -100
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    raise LessThanZeroError('my_num must be greater than 0')
__main__.LessThanZeroError
```

A programmer creates a custom exception type by creating a class that inherits from the built-in Exception class. The new class can contain a constructor, as shown above, that may accept an argument to be saved as an attribute. Alternatively, the class could have no constructor (and a "pass" statement might be used, since a class definition requires at least one statement). A custom exception class is typically kept bare, adding a minimal amount of functionality to keep track of information that an exception handler might need. Inheritance is discussed in detail elsewhere.

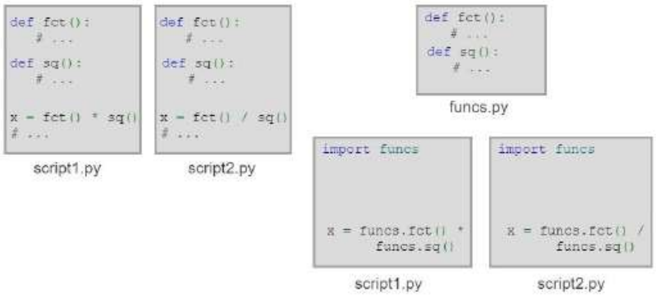
Good practice is to include "Error" at the end of a custom exception type's name, as in LessThanZeroError or MyError. Custom exception types are useful to track and handle the unique exceptions that might occur in a program's code. Many larger third-party and Python standard library modules use custom exception types.

Modules

Script Python code written in a file that is passed as input to the interpreter

Module a file containing Python code that can be imported and used by scripts, other modules, or the interactive interpreter

To **import** a module means to execute the code contained by the module, and make the definitions within that module available for use by the importing program



```
def fct():
    # ...
def sq():
    # ...

x = fct() * sq()
# ...
```

```
def fct():
    # ...
def sq():
    # ...

x = fct() / sq()
# ...
```

```
def fct():
    # ...
def sq():
    # ...
```

```
import funcs

x = funcs.fct() *
    funcs.sq()
```

```
import funcs

x = funcs.fct() /
    funcs.sq()
```

The functions can instead be defined in another file. The file can be imported as a 'module'.

Captions ^

1. A programmer writes scripts containing functions and code using those functions. Multiple scripts might define the same functions.
2. The functions can instead be defined in another file. The file can be imported as a 'module'.

A module's filename should end with ".py"; otherwise, the interpreter will not be able to import the module. The module_name item should match the filename of the module, but without the .py extension. Ex: If a programmer wants to import a module whose filename is HTTPServer.py, the import statement `import HTTPServer` would be used. Note that import statements are case-sensitive; thus, `import ABC` is distinct from `import aBc`.

The interpreter must also be able to find the module to import. The simplest solution is to keep modules in the same directory as the executing script; however, the interpreter can also search the computer's file system for the modules. Later material covers these search mechanisms.

Good practice is to place import statements at the top of a file. There are few useful instances of placing import statements in any location other than the top. The benefit of placing import statements at the top is that a reader of the program can quickly identify the modules required for the program to run. A module being required by another program is often called a ***dependency***.

Evaluating an import statement initiates the following process to load the module:

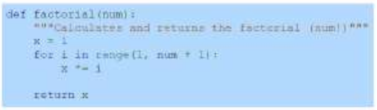
1. A check is conducted to determine whether the module has already been imported. If already imported, then the loaded module is used.
2. If not already imported, a new module object is created and inserted in sys.modules.
3. The code in the module is executed in the new module object's namespace.

A dictionary of the loaded modules is stored in sys.modules

A module object is simply a namespace that contains definitions from the module

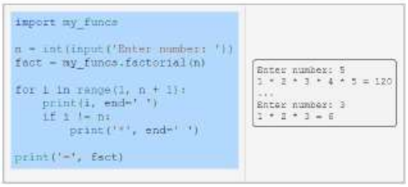
Once a module has been imported, the program can access the definitions of a module using attribute reference operations, e.g., `my_ip = HTTPServer.address` sets my_ip to address defined in HTTPServer.py. The definitions can also be overwritten, e.g., `HTTPServer.address = "www.yahoo.com"` binds address in HTTPServer to 'www.yahoo.com'. Note that such changes are temporary and restricted to the current executing Python instance. Ending the program and then re-importing the module would reload the original value of HTTPServer.address.

Figure 13.1.1: Contents of my_funcs.py.



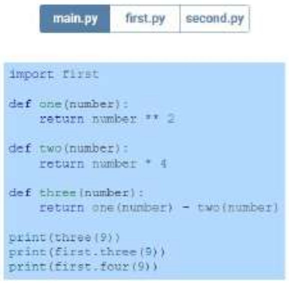
```
def factorial(num):
    """Calculates and returns the factorial (num!)"""
    x = 1
    for i in range(1, num + 1):
        x *= i

    return x
```

Figure 13.1.2: Using factorial from my_funcs.py.

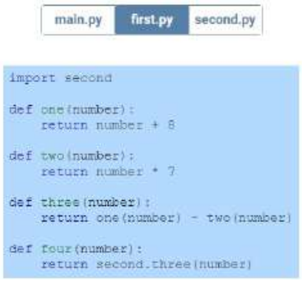


```
import my_funcs

n = int(input('Enter number: '))
fact = my_funcs.factorial(n)

for i in range(1, n + 1):
    print(i, end=' ')
    if i != n:
        print('*', end=' ')

print('=', fact)
```

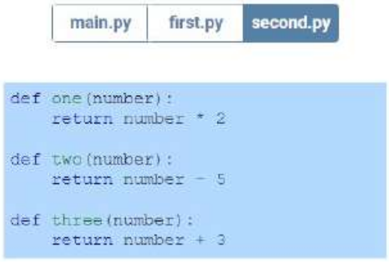
```
Enter number: 5
1 * 2 * 3 * 4 * 5 = 120
...
Enter number: 3
1 * 2 * 3 = 6
```



```
import first

def one(number):
    return number ** 2

def two(number):
    return number * 4

def three(number):
    return one(number) - two(number)

print(three(9))
print(first.three(9))
print(first.four(9))
```



```
import second

def one(number):
    return number + 8

def two(number):
    return number * 7

def three(number):
    return one(number) - two(number)

def four(number):
    return second.three(number)
```



```
def one(number):
    return number * 2

def two(number):
    return number - 5

def three(number):
    return number + 3
```

```
45
-46
12
```

Finding Modules

A built-in module is a module that comes pre-installed with Python, examples include sys, time, and math

A list of directories are contained by **sys.path**

The sys.path variable initially contains the following directories:

1. The directory of the executing script.
2. A list of directories specified by the environment variable PYTHONPATH.
3. The directory where Python is installed.

For simple programs, a module might simply be placed in the same directory. Larger projects might contain tens or hundreds of modules or use third-party modules located in different directories. In such cases, a programmer might set the environment variable ***PYTHONPATH*** in the operating system. An operating system ***environment variable*** is much like a variable in a Python script, except that an environment variable is stored by the computer's operating system and can be accessed by every program running on the computer. In Windows, a user can set the value of PYTHONPATH permanently through the control panel, or temporarily on a single instance of a command terminal (cmd.exe) using the command `set PYTHONPATH="c:\dir1;c:\other\directory"`.

Importing Specific Names from a Module

A programmer can specify names to import from a module by using the **from** keyword in an import statement

```
from module_name import name1, name2, ...
```

The program below imports names from the **hashlip** module, a Python standard library module that contains a number of algorithms for creating a secure **hash** of a text message

```
from hashlib import md5, sha1

text = input("Enter text to hash ('q' to quit): ")

while text != 'q':
    algorithm = input('Enter algorithm (md5/sha1): ')
    if algorithm == 'md5':
        output = md5(text.encode('utf-8'))
    elif algorithm == 'sha1':
        output = sha1(text.encode('utf-8'))
    else:
        output = 'Invalid algorithm selection'
    print('Hash value:', output.hexdigest())

    text = input("\nEnter text to hash ('q' to quit): ")
```

```
Enter text to hash ('q' to quit): Whether 'tis nobler in the mind to suffer...
Enter algorithm (md5/sha1): md5
Hash value: 5b39e6686305363a2d60a4162fe3d012

Enter text to hash ('q' to quit): ...the slings and arrows of outrageous fortune,...
Enter algorithm (md5/sha1): sha1
Hash value: 70c137974ad24691c1bb6cf8114aa2e3172ef910

Enter text to hash ('q' to quit): q
```

The hashlib library requires argument strings to md5 and sha1 be encoded; above, we encode the text using UTF-8 before passing to one of the hashing algorithms.

All names from a module can be imported directly by using a "*" character, as in the statement `from HTTPServer import *`. *A common error is to use the import * syntax in modules and scripts, which makes identification of dependencies and the origins of variables difficult for a reader of the code to understand. Good practice is to limit the use of import * syntax to interactive interpreter sessions.*

Table 13.3.1: 'import module' vs. 'from module import names'.

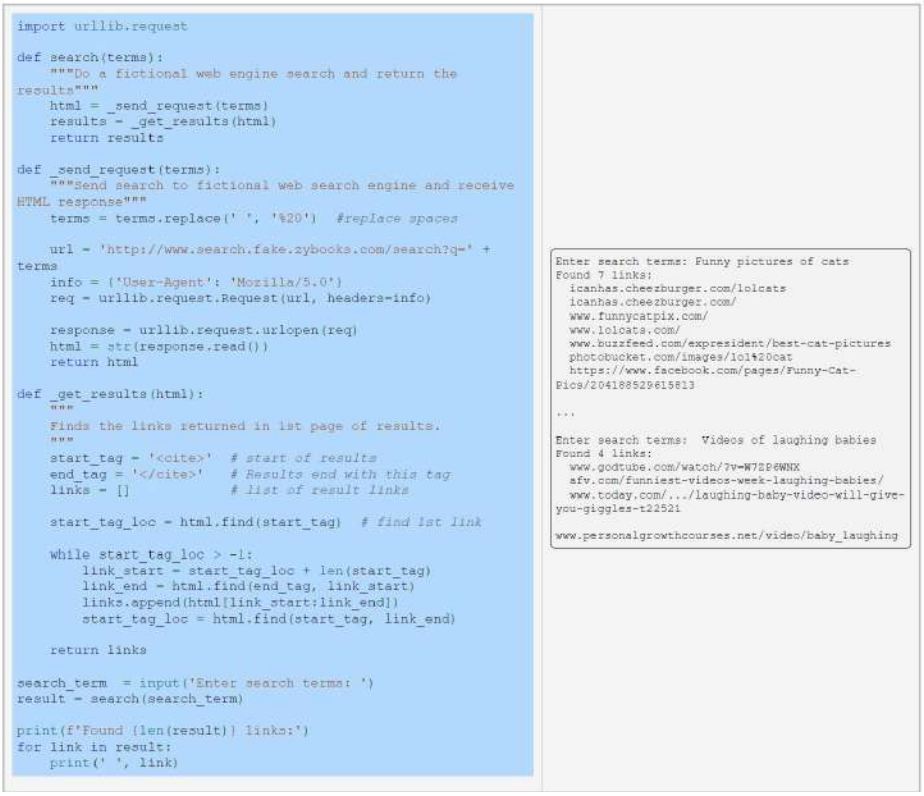
Description	Example import statement	Using imported names
Import an entire module	`import HTTPServer`	`print(HTTPServer.address)`
Import specific name from a module	`from HTTPServer import address`	`print(address)`

Executing Modules as Scripts

An import statement executes the code contained within the imported module. Thus, any statements in the global scope of a module, like printing or getting user input, will be executed when that module is imported. Execution of those statements may be an unintended side effect of the import. Commonly a programmer wants to treat a Python file as both a script executed by the interpreter and as an importable module. When used as an importable module, the file should not produce side effects when imported.

Ex: Consider the following Python file web_search.py, which contains functions for performing searches that "scrape" the results from a fictional web search engine, like Yahoo or Google. Executing the file as a script produces the following output:

Figure 13.4.1: web_search.py: Get the 1st page of results for a web search.

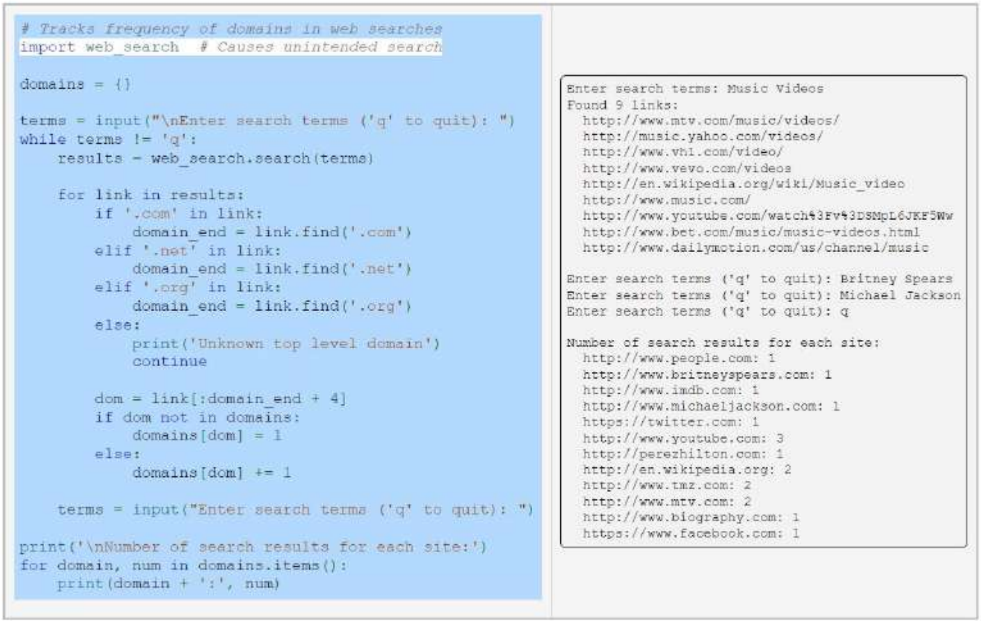


```
import urllib.request

def search(terms):
    """Do a fictional web engine search and return the
results"""
    html = _send_request(terms)
    results = _get_results(html)
    return results

def _send_request(terms):
    """Send search to fictional web search engine and receive
HTML response"""
    terms = terms.replace(' ', '%20')  #replace spaces

    url = 'http://www.search.fake.zybooks.com/search?q=' +
terms
    info = {'User-Agent': 'Mozilla/5.0'}
    req = urllib.request.Request(url, headers=info)

    response = urllib.request.urlopen(req)
    html = str(response.read())
    return html

def _get_results(html):
    """
    Finds the links returned in 1st page of results.
    """
    start_tag = '<cite>'  # start of results
    end_tag = '</cite>'   # Results end with this tag
    links = []            # list of result links

    start_tag_loc = html.find(start_tag)  # find 1st link

    while start_tag_loc > -1:
        link_start = start_tag_loc + len(start_tag)
        link_end = html.find(end_tag, link_start)
        links.append(html[link_start:link_end])
        start_tag_loc = html.find(start_tag, link_end)

    return links

search_term  = input('Enter search terms: ')
result = search(search_term)

print(f'Found {len(result)} links:')
for link in result:
    print(' ', link)
```

```
Enter search terms: Funny pictures of cats
Found 7 links:
  icanhas.cheezburger.com/lolcats
  icanhas.cheezburger.com/
  www.funnycatpix.com/
  www.lolcats.com/
  www.buzzfeed.com/expresident/best-cat-pictures
  photobucket.com/images/lol%20cat
  https://www.facebook.com/pages/Funny-Cat-
Pics/204188529615813

...

Enter search terms:  Videos of laughing babies
Found 4 links:
  www.godtube.com/watch/?v=W7ZP6WNX
  afv.com/funniest-videos-week-laughing-babies/
  www.today.com/.../laughing-baby-video-will-give-
you-giggles-t22521

www.personalgrowthcourses.net/video/baby_laughing
```

If another script imports web_search.py to use the search() function, the statements at the bottom of web_search.py will also execute. The domain_freq.py file below tracks the frequency of specific domains in search results; however, importing web_search.py causes a search and listing of each site to unintentionally occur, because that search is called at the global scope of web_search.py.

Figure 13.4.2: domain_freq.py: Importing web_search causes unintended search to occur.



```python
# Tracks frequency of domains in web searches
import web_search  # Causes unintended search

domains = {}

terms = input("\nEnter search terms ('q' to quit): ")
while terms != 'q':
    results = web_search.search(terms)

    for link in results:
        if '.com' in link:
            domain_end = link.find('.com')
        elif '.net' in link:
            domain_end = link.find('.net')
        elif '.org' in link:
            domain_end = link.find('.org')
        else:
            print('Unknown top level domain')
            continue

        dom = link[:domain_end + 4]
        if dom not in domains:
            domains[dom] = 1
        else:
            domains[dom] += 1

    terms = input("Enter search terms ('q' to quit): ")

print('\nNumber of search results for each site:')
for domain, num in domains.items():
    print(domain + ':', num)
```

```
Enter search terms: Music Videos
Found 9 links:
  http://www.mtv.com/music/videos/
  http://music.yahoo.com/videos/
  http://www.vh1.com/video/
  http://www.vevo.com/videos
  http://en.wikipedia.org/wiki/Music_video
  http://www.music.com/
  http://www.youtube.com/watch%3Fv%3DSMpL6JKF5Ww
  http://www.bet.com/music/music-videos.html
  http://www.dailymotion.com/us/channel/music

Enter search terms ('q' to quit): Britney Spears
Enter search terms ('q' to quit): Michael Jackson
Enter search terms ('q' to quit): q

Number of search results for each site:
  http://www.people.com: 1
  http://www.britneyspears.com: 1
  http://www.imdb.com: 1
  http://www.michaeljackson.com: 1
  https://twitter.com: 1
  http://www.youtube.com: 3
  http://perezhilton.com: 1
  http://en.wikipedia.org: 2
  http://www.tmz.com: 2
  http://www.mtv.com: 2
  http://www.biography.com: 1
  https://www.facebook.com: 1
```

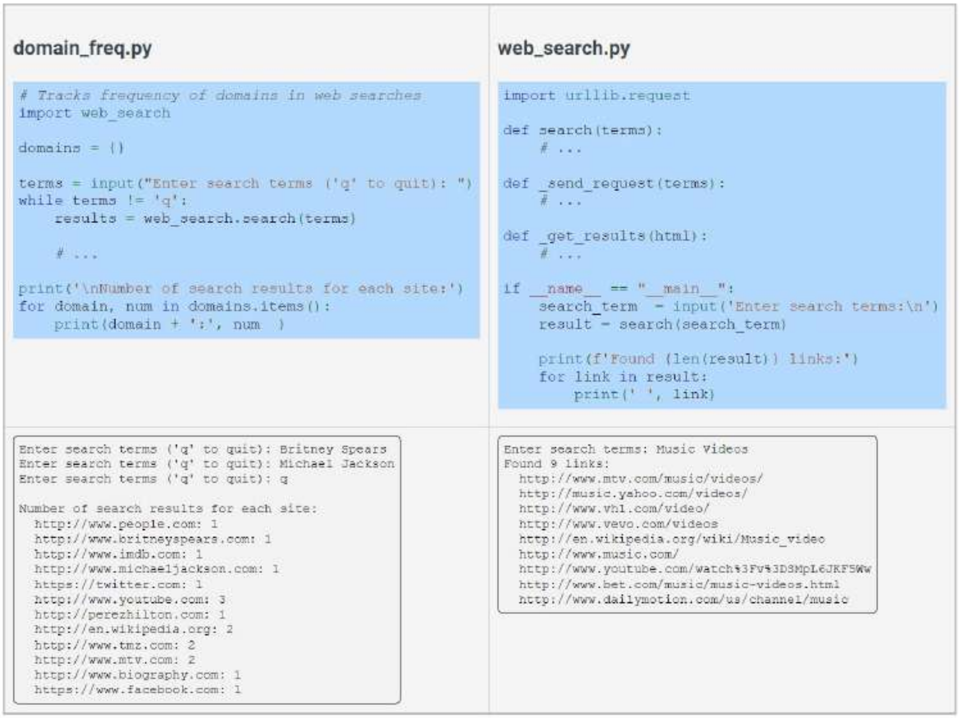
A file can better support execution as both a script and an importable module by utilizing the __name__ special name. ***__name__*** is a global string variable automatically added to every module that contains the name of the module. Ex: my_funcs.__name__ would have the value "my_funcs", and web_search.__name__ would have the value "web_search". (Note that __name__ has two underscores before name and two underscores after.) However, the value of __name__ for the executing script is always set to "__main__" to differentiate the script from imported modules. The following comparison can be performed:

If `if __name__ == "__main__"` is true, then the file is being executed as a script and the branch is taken. Otherwise, the file was imported and thus __name__ is equal to the module name, e.g., 'web_search'.

The contents of the branch typically include a user interface to functions or class definitions within the file. A user can execute the file as a script and interact with the user interface, or another script can import the file just to use the definitions. The web_search.py file is modified below to fix the unintentional search.

```python
if __name__ == "__main__":
    # File executed as a script
```

Figure 13.4.4: web_search.py modified to run as either script or module.



Each file below is executed as a script.

domain_freq.py

```python
# Tracks frequency of domains in web searches
import web_search

domains = {}

terms = input("Enter search terms ('q' to quit): ")
while terms != 'q':
    results = web_search.search(terms)

    # ...

print('\nNumber of search results for each site:')
for domain, num in domains.items():
    print(domain + ':', num  )
```

```
Enter search terms ('q' to quit): Britney Spears
Enter search terms ('q' to quit): Michael Jackson
Enter search terms ('q' to quit): q

Number of search results for each site:
  http://www.people.com: 1
  http://www.britneyspears.com: 1
  http://www.imdb.com: 1
  http://www.michaeljackson.com: 1
  https://twitter.com: 1
  http://www.youtube.com: 3
  http://perezhilton.com: 1
  http://en.wikipedia.org: 2
  http://www.tmz.com: 2
  http://www.mtv.com: 2
  http://www.biography.com: 1
  https://www.facebook.com: 1
```

web_search.py

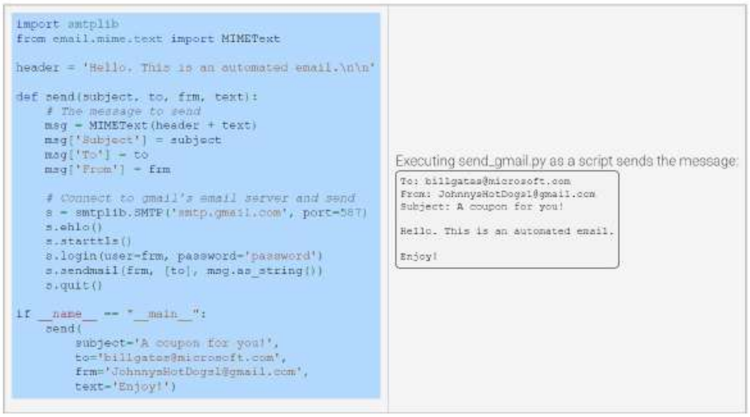
```python
import urllib.request

def search(terms):
    # ...

def _send_request(terms):
    # ...

def _get_results(html):
    # ...

if __name__ == "__main__":
    search_term  = input('Enter search terms:\n')
    result = search(search_term)

    print(f'Found {len(result)} links:')
    for link in result:
        print('  ', link)
```

```
Enter search terms: Music Videos
Found 9 links:
  http://www.mtv.com/music/videos/
  http://music.yahoo.com/videos/
  http://www.vh1.com/video/
  http://www.vevo.com/videos
  http://en.wikipedia.org/wiki/Music_video
  http://www.music.com/
  http://www.youtube.com/watch%3Fv%3DSMpL6JKF5Ww
  http://www.bet.com/music/music-videos.html
  http://www.dailymotion.com/us/channel/music
```

Reloading Modules

Sometimes a Python program imports a module, but then the source code of the imported module needs to be changed. Since modules are executed only once when imported, changing the module's source does not affect the running Python instance. Instead of restarting the entire Python program, the ***reload()*** function can be used to reload and re-execute the changed module. The reload() function is located in the importlib standard library module.

Figure 13.5.1: send_gmail.py: Sends a single email through gmail.



```python
import smtplib
from email.mime.text import MIMEText

header = 'Hello. This is an automated email.\n\n'

def send(subject, to, frm, text):
    # The message to send
    msg = MIMEText(header + text)
    msg['Subject'] = subject
    msg['To'] = to
    msg['From'] = frm

    # Connect to gmail's email server and send
    s = smtplib.SMTP('smtp.gmail.com', port=587)
    s.ehlo()
    s.starttls()
    s.login(user=frm, password='password')
    s.sendmail(frm, [to], msg.as_string())
    s.quit()

if __name__ == "__main__":
    send(
        subject='A coupon for you!',
        to='billgates@microsoft.com',
        frm='JohnnysHotDogs1@gmail.com',
        text='Enjoy!')
```

Executing send_gmail.py as a script sends the message:

```
To: billgates@microsoft.com
From: JohnnysHotDogs1@gmail.com
Subject: A coupon for you!

Hello. This is an automated email.

Enjoy!
```

Figure 13.5.2: send_coupons.py: Automates emails to loyal customers.

```
import os
from importlib import reload
import send_gmail

mod_time = os.path.getmtime(send_gmail.__file__)

emails = [  # Could be large list or stored in file
    'billgates@microsoft.com',
    'president@whitehouse.gov',
    'benedictxvi@vatican.va'
]

my_email = 'JohnnysHotDogs1@gmail.com'
subject = 'A coupon for you!'
text = ("As a loyal customer of Johnny's HotDogs, "
        "here is a coupon for 1 free bratwurst!")

for addr in emails:
    send_gmail.send(subject, addr, my_email, text)

    # Check if file has been modified
    last_mod = os.path.getmtime(send_gmail.__file__)
    if last_mod > mod_time:
        mod_time = last_mod
        reload(send_gmail)
```

If thousands of emails are being sent, the program should not be stopped because rerunning the program could cause duplicate emails to be sent to some users, and Johnny's HotDogs might annoy their customers. If Johnny wants to change the content of the header string in the send_gmail module without stopping the program, then the variable's value in send_gmail.py's *source code* can be updated and reloaded.

When send_coupons.py imports send_gmail, a global variable mod_time stores the time when send_gmail.py was last modified, using the os.path.getmtime() function. The ***__file__*** special name contains the path to a module in the computer file system, e.g., the value of send_gmail.__file__ might be 'C:\\Users\\Johnny\\send_gmail.py". A comparison is made to the original modification time at the end of the for loop – if the modification time is greater than the original, then the module's source code has been updated and the module should be reloaded.

Modifying the header string in send_gmail.py to *"This is an important message from Johnny"* while the program is running causes the module to be reloaded, which alters the contents of the emails.

Figure 13.5.4: Reloading modules doesn't affect attributes imported using 'from'.

```
from importlib import reload
import send_gmail
from send_gmail import header

print('Original value of header:', header)

print('\n(---- send_gmail.py source code edited ----)')

print('\nReloading send_gmail\n')
reload(send_gmail)

print('header:', header)
print('send_gmail.header:', send_gmail.header)
```

```
Original value of header: Hello. This is an automated email.

(---- send_gmail.py edited ----)

Reloading send_gmail.

header: Hello. This is an automated email.
send_gmail.header: Hello from Johnny's Hotdogs!
```

The reload function reloads a module in-place. When reload(send_gmail) returns, the namespace of the send_gmail module will contain updated definitions. The call to send_gmail.send() still accesses the same send_gmail module object, but the definition of send() will have been updated.

Importing attributes directly using "from", and then reloading the corresponding module, will *not* update the imported attributes.

Packages

A package is a directory that, when imported, gives access to all of the modules stored in the directory

The draw_scene.py script can import the ASCIIArt package using the following statement:

Figure 13.6.2: Importing the ASCIIArt package.

```
import ASCIIArt  # import ASCIIArt package
```

Need help? Feedback?

Specific modules or subpackages can be imported individually by specifying the path to the item, using periods in the import name. References to names within the imported module require that the entire path is specified.

Figure 13.6.3: Importing the canvas module.

```
import ASCIIArt.canvas  # imports the canvas.py module

ASCIIArt.canvas.draw_canvas()  # Definitions in canvas.py have full name specified
```

Need help? Feedback?

The *from* technique of importing also works with packages, allowing individual modules or subpackages to be directly imported into the global namespace. A benefit of this method is that higher level package names need not be specified.

Figure 13.6.4: Import cow module from figures subpackage.

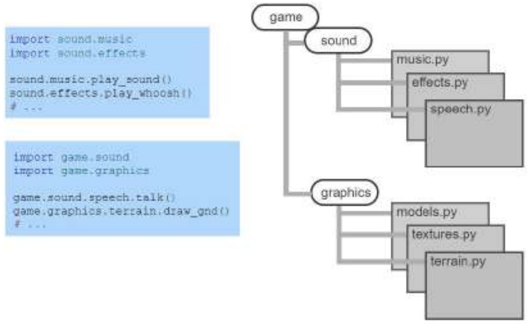
```
from ASCIIArt.figures import cow  # import cow module

cow.draw()  # Can omit ASCIIArt.figures prefix
```

Need help? Feedback?

Even individual names from a module can be imported, making that name directly available.

Figure 13.6.5: Import the draw function from the cow module.

```
from ASCIIArt.figures.cow import draw  # import draw() function

draw()  # Can omit ASCIIArt.figures.cow
```



```
import sound.music
import sound.effects

sound.music.play_sound()
sound.effects.play_whoosh()
# ...
```

```
import game.sound
import game.graphics

game.sound.speech.talk()
game.graphics.terrain.draw_gnd()
# ...
```

The 'game' package contains subpackages 'sound' and 'graphics'.

Captions

1. Packages, such as 'sound', contain subpackages, such as 'music' and 'effects'. Once subpackages are imported, modules and definitions in the subpackages are reached via dot notation.
2. The 'game' package contains subpackages 'sound' and 'graphics'.

To import a package, a programmer writes an import statement and gives the name of the directory where the package is located. To indicate that a directory is a package, the directory must include a file called __init__.py. The _init_.py file often is empty, but may include import statements that import subpackages or modules. The interpreter searches for the package in the directories listed in sys.path.

Figure 13.6.1: Directory structure.

```
draw_scene.py            Script that imports ASCIIArt package
ASCIIArt\                     Top-level package
        __init__.py
        canvas.py
        figures\              Subpackage for figures art
               __init__.py
               man.py
               cow.py
               ...
        buildings\            Subpackage for buildings art
               __init__.py
               barn.py
               house.py
               ...
```

When using syntax such as "import y.z", the last item z must be a package, a module, or a subpackage. In contrast, when using "from x.y import z", the last item z can also be a name from y, such as a function, class, or global variable.

Using packages helps to avoid module name collisions. For example, consider if another package called 3DGraphics also contained a module called canvas.py. Though both modules share a name, they are differentiated by the package that contains them, i.e., ASCIIArt.canvas is different from 3DGraphics.canvas. A programmer should take care when using the *from* technique of importing. *A common error is to overwrite an imported module with another package's identically named module.*

Standard Library

The Python standard library includes various utilities and tools for performing common program behaviors

Table 13.7.1: Some commonly used Python standard library modules.

Module name	Description	Documentation link
datetime	Creation and editing of dates and times objects	https://docs.python.org/3/library/datetime.html
random	Functions for working with random numbers	https://docs.python.org/3/library/random.html
copy	Create complete copies of objects	https://docs.python.org/3/library/copy.html
time	Get the current time, convert time zones, sleep for a number of seconds	https://docs.python.org/3/library/time.html
math	Mathematical functions	https://docs.python.org/3/library/math.html
os	Operating system informational and management helpers	https://docs.python.org/3/library/os.html
sys	System specific environment or configuration helpers	https://docs.python.org/3/library/sys.html
pdb	The Python interactive debugger	https://docs.python.org/3/library/pdb.html
urllib	URL handling functions, such as requesting web pages	https://docs.python.org/3/library/urllib.html

Figure 13.7.1: Using the datetime module.

The following program uses the datetime module to print the day, month, and year of a date that is a user-entered number of days in the future.

```
import datetime

# Create a new date object representing the current date (May 30, 2016)
today  = datetime.date.today()

days_from_now = int(input('Enter number of days from now: '))

# Create a new timedelta object that represents a difference in the
# number of days between dates.
day_difference = datetime.timedelta(days = days_from_now)

# Calculate new date
future_date = today + day_difference

print(days_from_now, 'days from now is', future_date.strftime('%B %d, %Y'))
```

```
Enter number of days from now: 30
30 days from now is June 29, 2016
```

Figure 13.7.2: Using the random module.

The following program uses the *random* module to implement a simple game where a user continues to draw from a deck of cards until an ace card is found.

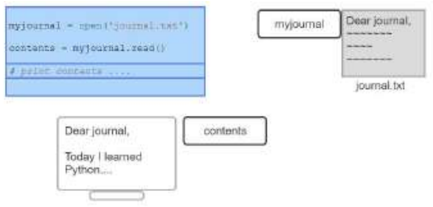
```
import random

# Create a shuffled card deck with 4 suites of cards 2-10, and face
cards
deck = [2, 3, 4, 5, 6, 7, 8, 9, 10, 'J', 'Q', 'K', 'A'] * 4
random.shuffle(deck)

num_drawn = 0
game_over = False
user_input = input("Press any key to draw a card ('q' to quit): ")
while user_input != 'q' and not game_over:

    # Draw a random card, and remove card from the deck
    card = random.choice(deck)
    deck.remove(card)
    num_drawn += 1

    print('\nCard drawn:', card)

    # Game is over if an ace was drawn
    if card == 'A':
        game_over = True
    else:
        user_input = input("Press any key to draw a card ('q' to
quit): ")

if user_input == 'q':
    print("\nGame was quit")
else:
    print(num_drawn, 'card(s) were drawn to find an ace.')
```

```
Press any key to draw a card ('q' to
quit): g
Card drawn: 10
Press any key to draw a card ('q' to
quit): g
Card drawn: 5
Press any key to draw a card ('q' to
quit): g
Card drawn: K
Press any key to draw a card ('q' to
quit): g
Card drawn: 9
Press any key to draw a card ('q' to
quit): g
Card drawn: A
5 card(s) were drawn to find an ace.
```

Files

Reading Files

A common programming task is to get input from a file using the built-in open() function rather than from a user typing on a keyboard



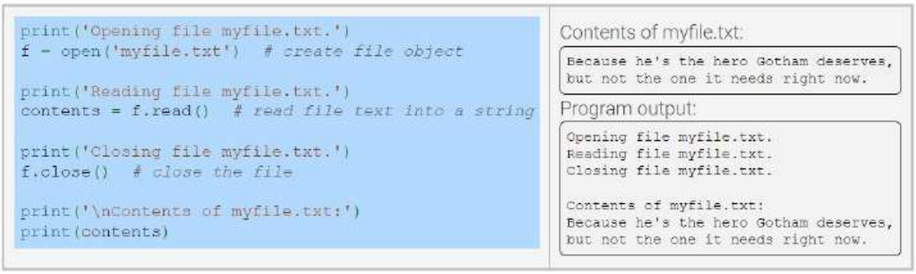
```
myjournal = open('journal.txt')
contents = myjournal.read()
# print contents ....
```

The open function creates a file object. The read function saves the content of the file as a string

Captions ^

1. The open function creates a file object. The read function saves the content of the file as a string.

Figure 14.1.1: Creating a file object and reading text.



```
print('Opening file myfile.txt.')
f = open('myfile.txt')  # create file object

print('Reading file myfile.txt.')
contents = f.read()  # read file text into a string

print('Closing file myfile.txt.')
f.close()  # close the file

print('\nContents of myfile.txt:')
print(contents)
```

Contents of myfile.txt:

```
Because he's the hero Gotham deserves,
but not the one it needs right now.
```

Program output:

```
Opening file myfile.txt.
Reading file myfile.txt.
Closing file myfile.txt.

Contents of myfile.txt:
Because he's the hero Gotham deserves,
but not the one it needs right now.
```

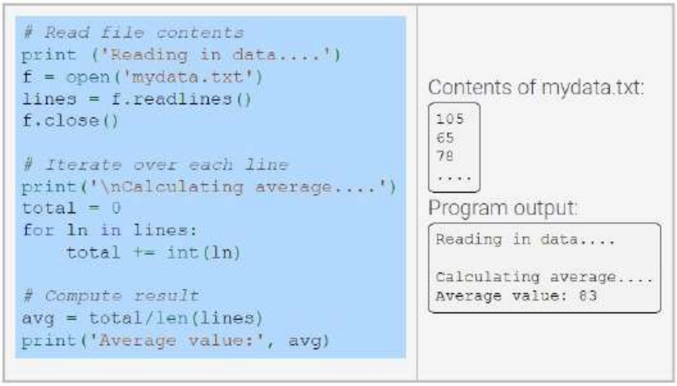
The open() built-in function requires a single argument that specifies the path to the file. Ex: `open('myfile.txt')` opens myfile.txt located in the same directory as the executing script. Full path names can also be specified, as in `open('C:\\Users\\BWayne\\tax_return.txt')`. The ***file.close()*** method closes the file, after which no more reads or writes to the file are allowed.

The most common methods to read text from a file are file.read() and file.readlines(). The ***file.read()*** method returns the file contents as a string. The ***file.readlines()*** method returns a list of strings, where the first element is the contents of the first line, the second element is the contents of the second line, and so on. Both methods can be given an optional argument that specifies the number of bytes to read from the file. Each method stops reading when the end-of-file (***EOF***) is detected, which indicates no more data is available.

A third method, file.readline(), returns a single line at a time, which is useful when dealing with large files where the entire file contents may not fit into the available system memory.

Figure 14.1.2: Calculating the average of data values stored in a file.

The file "mydata.txt" contains 100 integers, each on its own line:



```
# Read file contents
print ('Reading in data....')
f = open('mydata.txt')
lines = f.readlines()
f.close()

# Iterate over each line
print('\nCalculating average....')
total = 0
for ln in lines:
    total += int(ln)

# Compute result
avg = total/len(lines)
print('Average value:', avg)
```

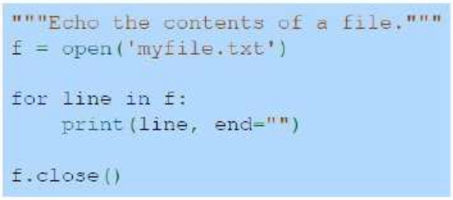
Contents of mydata.txt:

```
105
65
78
....
```

Program output:

```
Reading in data....

Calculating average....
Average value: 83
```

Figure 14.1.3: Iterating over the lines of a file.



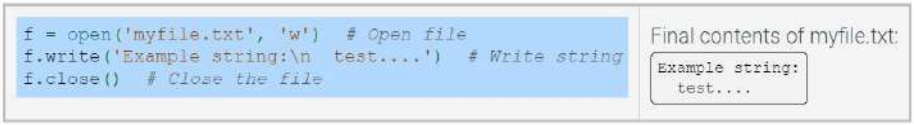
```
"""Echo the contents of a file."""
f = open('myfile.txt')

for line in f:
    print(line, end="")

f.close()
```

```
my_file = open('readme.txt')

contents = my_file.read(500)

# ....
```

```
my_file = open('readme.txt')

lines = my_file.readlines()

print( lines[1] )

# ....
```

Writing Files

The file.write() method writes a string argument to a file



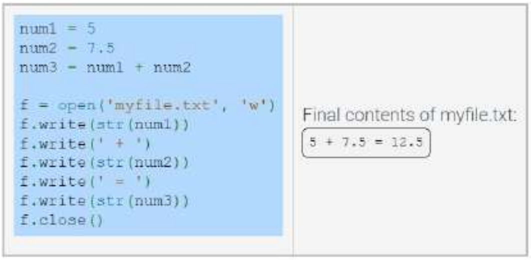
```
f = open('myfile.txt', 'w')  # Open file
f.write('Example string:\n  test....')  # Write string
f.close()  # Close the file
```

Final contents of myfile.txt:

```
Example string:
  test....
```

The write() method accepts a string argument only. Integers and floating-point values must first be converted using str(), as in `f.write(str(5.75))`.

Figure 14.2.2: Numeric values must be converted to strings.



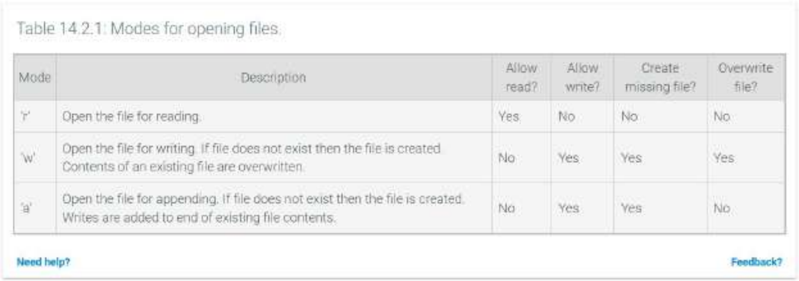
```
num1 = 5
num2 = 7.5
num3 = num1 + num2

f = open('myfile.txt', 'w')
f.write(str(num1))
f.write(' + ')
f.write(str(num2))
f.write(' = ')
f.write(str(num3))
f.close()
```

Final contents of myfile.txt:

```
5 + 7.5 = 12.5
```

The below table lists common file modes:

Table 14.2.1: Modes for opening files.



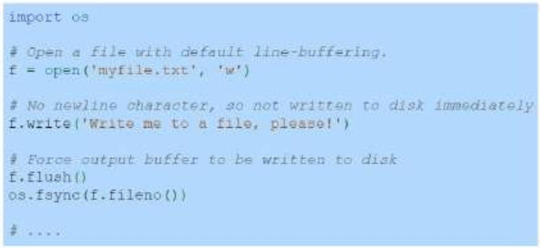
Mode	Description	Allow read?	Allow write?	Create missing file?	Overwrite file?
'r'	Open the file for reading.	Yes	No	No	No
'w'	Open the file for writing. If file does not exist then the file is created. Contents of an existing file are overwritten.	No	Yes	Yes	Yes
'a'	Open the file for appending. If file does not exist then the file is created. Writes are added to end of existing file contents.	No	Yes	Yes	No

Need help? Feedback?

A mode indicates how a file is opened

- Read mode 'r' opens a file for reading. If the file is missing, then an error will occur.
- Write mode 'w' opens a file for writing. If the file is missing, then a new file is created. Contents of any existing file are overwritten.
- Append mode 'a' opens a file for writing. If the file is missing, then a new file is created. Writes to the file are appended to the end of an existing file's contents.

Figure 14.2.3: Using flush() to force an output buffer to write to disk.



```
import os

# Open a file with default line-buffering.
f = open('myfile.txt', 'w')

# No newline character, so not written to disk immediately
f.write('Write me to a file, please!')

# Force output buffer to be written to disk
f.flush()
os.fsync(f.fileno())

# ....
```

The flush() file method can be called to force the interpreter to flush the output buffer to disk

Interacting with File Systems

The Python standard library's OS module provides an interface to operating system function calls and is thus a critical piece of a Python programmer's toolbox

A programmer should consider the ***portability*** of a program across different operating systems to avoid scenarios where the program behaves correctly on the programmer's computer but crashes on another. Portability must be considered when reading and writing files outside the executing program's directory since file path representations often differ between operating systems. For example, on Windows the path to a file is represented as "subdir\\bat_mobile.jpg", but on a Mac is 'subdir/bat_mobile.jpg'. The character between directories, e.g., "\\"or "/", is called the ***path separator***, and using the incorrect path separator may result in that file not being found.[1]

A common error is to reduce a program's portability by hardcoding file paths as string literals with operating system specific path separators. To help reduce such errors, good practice is to use the os.path module, which contains many portable functions for handling file paths. One of the most useful functions is os.path.join(), which concatenates the arguments using the correct path separator for the current operating system. Instead of writing the literal `path = "subdir\\bat_mobile.jpg"`, a programmer should write `path = os.path.join('subdir', 'bat_mobile.jpg')`, which will result in `"subdir\\bat_mobile.jpg"` on Windows and `"subdir/bat_mobile.jpg"` on Linux/Mac.

Figure 14.3.1: Using os.path.join() to create a portable file path string.

The program below echoes the contents of logs stored in a hierarchical directory structure organized by date, using the os.path module to build a file path string that is portable across operating systems.

```
import os
import datetime

curr_day = datetime.date(1997, 8, 29)

num_days = 30
for i in range(num_days):
    year = str(curr_day.year)
    month = str(curr_day.month)
    day = str(curr_day.day)

    # Build path string using current OS path separator
    file_path = os.path.join('logs', year, month, day, 'log.txt')

    f = open(file_path, 'r')

    print(f'{file_path}: {f.read()}')
    f.close()

    curr_day = curr_day + datetime.timedelta(days=1)
```

Output on Windows:

```
logs\1997\8\29\log.txt:  # ....
logs\1997\8\30\log.txt:  # ....
# ....
logs\1997\9\28\log.txt:  # ....
```

Output on Linux:

```
logs/1997/8/29/log.txt:  # ....
logs/1997/8/30/log.txt:  # ....
# ....
logs/1997/9/28/log.txt:  # ....
```

On Windows systems, when using os.path.join() with a full path such that the first argument is a drive letter (e.g., 'C:' or 'D:'), the separator must be included with the drive letter. For example, `os.path.join('C:\\', 'subdir1', 'myfile.txt')` returns the string "C:\\subdir1\\myfile.txt".

The inverse operation, splitting a path into individual tokens, can be done using the str.split() method. Ex: `tokens = 'C:\\Users\\BWayne\\tax_return.txt'.split(os.path.sep)` returns ['C:', 'Users', 'BWayne', 'tax_return.txt']. ***os.path.sep*** stores the path separator for the current operating system.

Figure 14.3.2: Directory structure organized by date.

```
logs/
    2009/
        April/
                1/
                  log.txt
                  words.doc
        January/
                15/
                  log.txt
                21/
                  log.txt
                  temp23.pdf
                24/
                  presentation.ppt
    2010/
        March/
              3/
               log.txt
              7/
               music.mp3
```

The os.walk() function 'walks' a directory tree like the one above, visiting each subdirectory in the specified path

Figure 14.3.3: Walking a directory tree.

```
import os

year = input('Enter year: ')
path = os.path.join('logs', year)
print()

for dirname, subdirs, files in os.walk(path):
    print(dirname, 'contains subdirectories:', subdirs, end=' ')
    print('and the files:', files)
```

```
Enter year:2009

logs\2009 contains subdirectories: ['April', 'January'] and the files: []
logs\2009\April contains subdirectories: ['1'] and the files: []
logs\2009\April\1 contains subdirectories: [] and the files: ['log.txt', 'words.doc']
logs\2009\January contains subdirectories: ['15', '21', '24'] and the files: []
logs\2009\January\15 contains subdirectories: [] and the files: ['log.txt']
logs\2009\January\21 contains subdirectories: [] and the files: ['log.txt', 'temp23.pdf']
logs\2009\January\24 contains subdirectories: [] and the files: ['presentation.ppt']
```

The os.walk() function is used as the iterable object in a for loop that yields a 3-tuple for each iteration.[2] The first item *dirname* contains the path to the current directory. The second item *subdirs* is a list of all the subdirectories of the current directory. The third item *files* is a list of all the files residing in the current directory.

A programmer might use os.walk() when searching for specific files within a directory tree, and the exact path is unknown. Another common task is to filter files based on their file extensions (.pdf, .txt, etc.), which are a convention used to indicate the type of data that a file holds.

(*1) Unix-based operating systems, like Linux and Mac OS X, will not recognize paths using the windows "\\" separator. Generally, Windows recognizes both "/" and "\\". The double backslash "\\" is the escape sequence to represent a single backslash within a String.

(*2) os.walk() actually returns a special object called a generator, which is discussed elsewhere.

The os and os.path modules contain other helpful functions, such as checking if a given path is a directory or a file, getting the size of a file, obtaining a file's extension (e.g., .txt, .doc, .pdf), creating and deleting directories, etc. Some of the most commonly used functions are listed below:

- os.path.split(path) – *Splits a path into a 2-tuple (head, tail), where tail is the last token in the path string and head is everything else.*

```
import os
p = os.path.join('C:\\', 'Users', 'BWayne', 'batsuit.jpg')
print(os.path.split(p))
```

```
('C:\\Users\\BWayne', 'batsuit.jpg')
```

- os.path.exists(path) – *Returns True if path exists, else returns False.*

```
import os
p = os.path.join('C:\\', 'Users', 'BWayne', 'batsuit.jpg')
if os.path.exists(p):
    print('Suit up....')
else:
    print('The Lamborghini then?')
```

If file exists:

```
Suit up....
```

If file does not exist:

```
The Lamborghini then?
```

- os.path.isfile(path) – *Returns True if path is an existing file, and false otherwise (e.g., path is a directory).*

```
import os
p = os.path.join('C:\\', 'Users', 'BWayne', 'bat_chopper')
if os.path.isfile(p):
    print('Found a file....')
else:
    print('Not a file....')
```

If path is a file:

```
Found a file....
```

If path is not a file:

```
Not a file....
```

- os.path.getsize(path) – *Returns the size in bytes of path.*

```
import os
p = os.path.join('C:\\', 'Users', 'BWayne', 'batsuit.jpg')
print('Size of file:', os.path.getsize(p), 'bytes')
```

```
Size of file: 65544 bytes
```

Binary Data

Some files consist of data stored as a sequence of bytes, known as ***binary data***, that is not encoded into human-readable text using an encoding like ASCII or UTF-8. Images, videos, and PDF files are examples of the types of files commonly stored as binary data. Opening such a file with a text editor displays text that is incomprehensible to humans because the text editor attempts to encode raw byte values into readable characters.

A ***bytes object*** is used to represent a sequence of single byte values, such as binary data read from a file. Bytes objects are immutable, just like strings, meaning the value of a bytes object cannot change once created. A byte object can be created using the ***bytes()*** built-in function:

- *bytes('A text string', 'ascii')* – creates a sequence of bytes by encoding the string using ASCII.
- *bytes(100)* – creates a sequence of 100 bytes whose values are all 0.
- *bytes([12, 15, 20])* – creates a sequence of 3 bytes with values from the list.

Alternatively, a programmer can write a bytes literal, similar to a string literal, by prepending a 'b' prior to the opening quote:

Figure 14.4.1: Creating a bytes object using a bytes literal.

```
my_bytes = b'This is a bytes literal'

print(my_bytes)
print(type(my_bytes))
```

```
b'This is a bytes literal'
<class 'bytes'>
```

Figure 14.4.2: Byte string literals.

```
print(b'123456789 is the same as \x31\x32\x33\x34\x35\x36\x37\x38\x39')
```

```
b'123456789 is the same as 123456789'
```

Programs can also access files using a ***binary file mode*** by adding a "b" character to the end of the mode string in a call to open(), as in `open('myfile.txt', 'rb')`. When using binary file mode "b" on a Windows computer, newline characters "\n" in the file are *not* automatically mapped to the Windows format "\r\n". In normal text mode, i.e., when not using the "b" binary mode, Python performs this translation of line-endings as a helpful feature, easing compatibility issues between Windows and other operating systems. In binary mode, the translation is not done because inserting additional characters would corrupt the binary data. On non-Windows systems, newline characters are not translated when using binary mode.

When a file is opened using a binary mode, the file.read() method returns a bytes object instead of a string. Also, the file.write() method expects a bytes argument.

Figure 14.4.3: Inspecting the binary contents of an image file.

```
f = open('ball.bmp', 'rb')  # Open in binary mode using 'b'

# Read image binary data
contents = f.read()

print('Contents of ball.bmp:\n')
print(contents)

f.close()
```

Abbreviated output:

```
Contents of ball.bmp:

b'BMb\xe6\x00\x00\x00\x00\x00\x006\x04\x00\x00(\x00\x00\x00,\x01\x00\x00
\xc1\x00\x00\x00\x01\x00\x08\x00\x00\x00\x00\x00,\xe2\x00\x00\xc4'
```

The `print(contents)` statement displays the value of contents, converting each byte to human-readable character if that byte's value is a readable ASCII character (less than 128). The first portion of the file's contents is shown in the output, though the file portion is abbreviated because the image contains about 27,000 bytes. Note how the first 14 bytes of the bitmap file is "BMb\xe6\x00\x00\x00\x00\x00\x006\x04\x00\x00". This sequence constitutes the ***header*** of the binary file, which describes the bitmap's contents. The first 2 bytes "BM" indicates the type of bitmap. The following 4 bytes "b\xe6\x00\x00" indicates the size of the bitmap. The sequence "6\x04\x00\x00" indicates where in the file the RGB (red-green-blue) values for each pixel in the image are stored.

Example 14.4.1: Altering a BMP image file.

The following program reads in ball.bmp, overwrites a portion of the image with new pixel colors, and creates a new image file. Download the above image (click the link, "ball.bmp", above the image), and then run the program on your own computer, creating a new, altered version of ball.bmp. Try changing the alterations made by the program to get different colors.

```
import struct

ball_file = open('ball.bmp', 'rb')
ball_data = ball_file.read()
ball_file.close()

# BMP image file format stores location
# of pixel RGB values in bytes 10-14
pixel_data_loc = ball_data[10:14]

# Converts byte sequence into integer object
pixel_data_loc = struct.unpack('<L', pixel_data_loc)[0]

# Create sequence of 3000 red, green, and yellow pixels each
new_pixels = b'\x01'*3000 + b'\x02'*3000 + b'\x03'*3000

# Overwrite pixels in image with new pixels
new_ball_data = ball_data[:pixel_data_loc] + \
                new_pixels + \
                ball_data[pixel_data_loc + len(new_pixels):]

# Write new image
new_ball_file = open('new_ball.bmp', 'wb')
new_ball_file.write(new_ball_data)
new_ball_file.close()
```

The struct module is a commonly used Python standard library module for packing values into sequences of bytes into values (like integers and strings). The struct.pack() function packs values such as strings and integers into sequences of bytes:

Figure 14.4.4: Packing values into byte sequences.

```
import struct

print('Result of packing 5:', end=' ')
print(struct.pack('>h', 5))

print('Result of packing 256:', end=' ')
print(struct.pack('>h', 256))

print('Result of packing 5 and 256:', end=' ')
print(struct.pack('>hh', 5, 256))
```

```
Result of packing 5: b'\x00\x05'
Result of packing 256: b'\x01\x00'
Result of packing 5 and 256: b'\x00\x05\x01\x00'
```

The first argument to struct.pack() is a format string that describes how the following arguments should be converted into bytes. The "<" character indicates the **byte-order**, or endianness, of the conversion, which determines whether the most-significant or least-significant byte is placed first in the byte sequence. ">" places the most-significant byte first (big-endian), and "<" sets the least-significant byte first. The "h" character in the format strings above describe the type of object being converted, which most importantly determines how many bytes are used when packing the value. "h" describes the value being converted as a 2-byte integer; other common format characters are "b" for a 1-byte integer, "I" for a 4-byte integer, and "s" for a string. Explore the links at the end of the section for more information on the struct module.

The ***struct.unpack()*** module performs the reverse operation of struct.pack(), unpacking a sequence of bytes into a new object. Unpacking always returns a tuple with the results, even if only unpacking a single value:

Figure 14.4.5: Unpacking values from byte sequences.

The following code uses the repr() function, which returns a string version of an object.

```
import struct


print('Result of unpacking', repr('\x00\x05') + ":", end=' ')
print(struct.unpack('>h', b'\x00\x05'))


print('Result of unpacking', repr('\x01\x00') + ":", end=' ')
print(struct.unpack('>h', b'\x01\x00'))


print('Result of unpacking', repr('\x00\x05\x01\x00') + ":", end='
')
print(struct.unpack('>hh', b'\x00\x05\x01\x00'))
```

```
Result of unpacking '\x00\x05': (5,)
Result of unpacking '\x01\x00': (256,)
Result of unpacking '\x00\x05\x01\x00': (5,
256)
```

Command-line Arguments and Files

Figure 14.5.1: Using command-line arguments to specify the name of an input file.

```
import sys
import os

if len(sys.argv) != 2:

    print(f'Usage: {sys.argv[0]} input_file')
    sys.exit(1)   # 1 indicates error


print(f'Opening file {sys.argv[1]}.')

if not os.path.exists(sys.argv[1]):  # Make sure file exists
    print('File does not exist.')
    sys.exit(1)   # 1 indicates error

f = open(sys.argv[1], 'r')

# Input files should contain two integers on separate lines

print('Reading two integers.')
num1 = int(f.readline())
num2 = int(f.readline())


print(f'Closing file {sys.argv[1]}')
f.close()  # Done with the file, so close it


print(f'\nnum1: {num1}')

print(f'num2: {num2}')

print(f'num1 + num2: {num1 + num2}')
```

myfile1.txt:
```
5
10
```

myfile2.txt:
```
-34
7
```

```
>python my_script.py myfile1.txt
Opening file myfile1.txt.
Reading two integers.
Closing file myfile1.txt

num1: 5
num2: 10
num1 + num2: 15


>python my_script.py myfile2.txt
Opening file myfile2.txt.
Reading two integers.
Closing file myfile2.txt

num1: -34
num2: 7
num1 + num2: -27


>python my_script.py myfile3.txt
Opening file myfile3.txt.
File does not exist.
```

The 'with' Statement

A with statement can be used to open a file, execute a block of statements, and automatically close the file when complete

```
with open('myfile.txt', 'r') as myfile:
    # Statement-1
    # Statement-2
    # ....
    # Statement-N
```

Above, the file object returned by open() is bound to myfile. When the statements in the block complete, then myfile is closed. The with statement creates a ***context manager***, which manages the usage of a resource, like a file, by performing setup and teardown operations. For files, the teardown operation is automatic closure. Other context managers exist for other resources, and new context managers can be written by a programmer, but is out of scope for this material.

Forgetting to close a file can sometimes cause problems. For example, a file opened in write mode cannot be written to by other programs. *Good practice is to use a with statement when opening files, to guarantee that the file is closed when no longer needed.*

Figure 14.6.1: Using the with statement to open a file.

```
print('Opening myfile.txt')

# Open a file for reading and appending
with open('myfile.txt', 'r+') as f:
    # Read in two integers
    num1 = int(f.readline())
    num2 = int(f.readline())

    product = num1 * num2

    # Write back result on own line
    f.write('\n')
    f.write(str(product))

# No need to call f.close() - f closed automatically
print('Closed myfile.txt')
```

Comma Separated Values Files

A comma separated values (csv) file is a simple text-based file format that uses commas to separate data items, called fields

Contents of a csv file.

```
name,hw1,hw2,midterm,final
Petr Little,9,8,85,78
Sam Tarley,10,10,99,100
Joff King,4,2,55,61
```

Each line in the file above represents a row, and fields between commas on each row are in the same column as fields in the same position in each line. For example, the first row contains the items "name", "hw1", "hw2", "midterm", and "final"; the second row contains "Petr Little", "9", "8", "85" and "78". The first column contains "name", "Petr Little", "Sam Tarley", and "Joff King"; the second column contains "hw1", "9", "10", and "4".

The Python standard library ***csv module*** can be used to help read and write files in the csv format. To read a file using the csv module, a program must first create a *reader* object, passing a file object created via *open*. The reader object is an iterable – iterating over the reader using a for loop returns each row of the csv file as a list of strings, where each item in the list is a field from the row.

Figure 14.7.2: Reading each row of a csv file.

```
import csv

with open('grades.csv', 'r') as csvfile:
    grades_reader = csv.reader(csvfile, delimiter=',')

    row_num = 1
    for row in grades_reader:
        print(f'Row #{row_num}:', row)
        row_num += 1
```

Echoed file contents:

```
Row #1: ['name', 'hw1', 'hw2', 'midterm', 'final']
Row #2: ['Petr Little', '9', '8', '85', '78']
Row #3: ['Sam Tarley', '10', '10', '99', '100']
Row #4: ['Joff King', '4', '2', '55', '61']
```

The optional delimiter argument in the csv.reader() function specifies the character used in the csv file to separate fields; by default a comma is used. In some cases, the field itself may contain a comma – for example if the name of a student was specified as "lastname,firstname". In such a case, the csv file might instead use semicolons or some other rare character, e.g., Little, Petr;9;8;85;78. An alternative to changing the delimiter is to use quotes around the item containing the comma, e.g., "Little, Petr",9,8,85,78.

If the contents of the fields are numeric, then a programmer may want to convert the strings to integer or floating-point values to perform calculations with the data. The example below reads each row using a reader object and calculates a student's final score in the class:

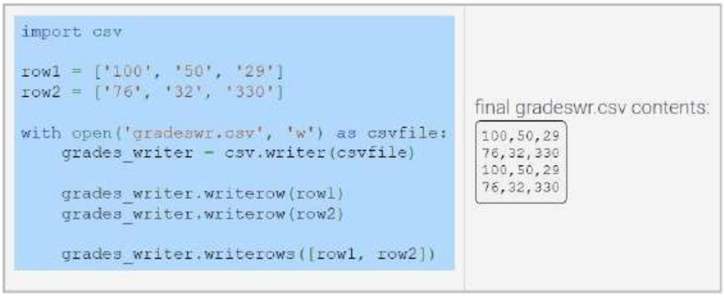
Figure 14.7.3: Using csv file contents to perform calculations.

```
import csv

# Dictionary that maps student names to a list of scores
grades = {}

# Use with statement to guarantee file closure
with open('grades.csv', 'r') as csvfile:
    grades_reader = csv.reader(csvfile, delimiter=',')

    first_row = True
    for row in grades_reader:
        # Skip the first row with column names
        if first_row:
            first_row = False
            continue

        ## Calculate final student grade ##

        name = row[0]

        # Convert score strings into floats
        scores = [float(cell) for cell in row[1:]]

        hw1_weighted = scores[0]/10 * 0.05
        hw2_weighted = scores[1]/10 * 0.05
        mid_weighted = scores[2]/100 * 0.40
        fin_weighted = scores[3]/100 * 0.50

        grades[name] = (hw1_weighted + hw2_weighted +
                        mid_weighted + fin_weighted) * 100

for student, score in grades.items():
    print(f'{student} earned {score:.1f}%')
```

```
Petr Little earned 81.5%
Sam Tarley earned 99.6%
Joff King earned 55.5%
```

A programmer can also use the csv module to write text into a csv file, using a *writer* object. The writer object's *writerow()* and *writerows* methods can be used to write a list of strings into the file as one or more rows.

Figure 14.7.4: Writing rows to a csv module.



```python
import csv

row1 = ['100', '50', '29']
row2 = ['76', '32', '330']

with open('gradeswr.csv', 'w') as csvfile:
    grades_writer = csv.writer(csvfile)

    grades_writer.writerow(row1)
    grades_writer.writerow(row2)

    grades_writer.writerows([row1, row2])
```

final gradeswr.csv contents:

```
100,50,29
76,32,330
100,50,29
76,32,330
```