python™ Programming Language

Python 3.x

Working with Python

Using Python Code

- **Versatile**
  - It works on Windows, Mac, and Linux and supports object-oriented and structured programming.
- **High level**
  - It is interpreted and uses natural language to present abstract programming functions.
- **Dynamically typed**
  - Its variable types change based on the data they hold rather than being declared statically on creation.
- **Many available libraries**
  - It is open source and easily accessible, so there are many modules containing predefined definitions that you can use to add easy functionality.
- **No backward compatibility**
  - Python 3.x is incompatible with Python 2.x code.

Importing Modules

Modules contain many valuable built-in definitions. Using these can save time and provide additional options to structure your program.

- `import x`
  - This gets the *x* module; you may access its functions by accessing the module.
    - `import math`
      `print(math.factorial(5))`
- `import x as y`
  - This gets the *x* module, but you may refer to it as *y* when accessing its functions.
    - `import math as M`
      `print(M.factorial(5))`
- `from x import function1`
  - This gets function1 from the *x* module; you may access it directly.
    - `from math import factorial`
      `print(factorial(5))`
- `from x import *`
  - This gets all functions from the *x* module that do not start with an underscore; you may access them directly as if you imported each one.
    - `from math import *`
      `print(factorial(5))`
  - **Note:** Using this method is inadvisable, because imported definitions may conflict with other definitions. It is better to import only the ones that you need to complete your project.
- `dir(x)`
  - This lists all definitions available in the *x* module.
    - `print(dir(math))`

Scope (Indentation)

The scope of a program defines when content starts and stops being accessible. When the scope changes, any items created within the new scope are inaccessible from prior scopes. This means that variables created within the scope of functions or statements cease being accessible when those functions or statements end.

- **Global scope**
  - This is accessible by the whole program and will persist as long as the program continues to run.
- **Meaningful indentation**
  - In Python, indentation, rather than a special set of characters, delineates change in scope. The further an item is indented from the left, the narrower its scope is.
  - Each segment of indentation is equivalent to four spaces, and tabs should not be used for indentation.

EX:

```
levelone = "global scope"

def function():
    leveltwo = "this function's scope"
    if True:
        levelthree = "this statement's scope"
        conclusionone = "I can change levelone and leveltwo from here"
    conclusiontwo = "levelthree and conclusionone no longer exist in this scope"

conclusionthree = "only levelone exists at this scope"
```

Naming Conventions

There are some naming conventions that you can follow to make your code more accessible to other Python developers, and some you need to be aware of to prevent naming conflicts.

- **Uppercase vs. lowercase**
  - Class names should be uppercase; all other names should be lowercase.
- **Leading underscores**
  - A single underscore before a name acts as a weak form of privatization. Function names that begin with one will not be imported when using the `from x import *` statement.
  - A double underscore before a name in a class mangles it, preventing subclasses from having conflicts with that name.
- **Trailing underscores**
  - A single underscore after a name is used to avoid naming conflicts, such as when using a reserved keyword for a name (e.g., `if_` or `def_`).
  - A double underscore after a name is used in conjunction with a double underscore before a name to indicate a name with a special function in Python (e.g., `__init__`).

Reserved Keywords

Some words hold a special meaning or function in Python. You cannot use these words to name variables, functions, or classes.

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

Comments

When creating a program, it can help other developers if you include comments that explain things in plain language.

- **Triple quotation marks**
  - When creating a program, method, function, or class, it is valuable to include a docstring that briefly explains the purpose. Do this directly below the declaration and enclose it in triple quotation marks.

EX:

```
def describedocstring():
    """You may use three single or double quotation marks to enclose this documentation"""
    print("docstrings tell users the purpose of objects!")
```

- **Pound sign**
  - You can include comments by using the # sign when describing the how, why, or what of your code. Everything on the line following the # will not be read by the interpreter and is there for future code readers.

EX:

```
#Comments can appear at the beginning of a line
print("Comments") #Or after code
```

Remember	
Do	**Do not**
Include comments for complicated or unintuitive functions	Repeat your code in your comments
Use docstrings for public functions	Comment excessively
Use a docstring to describe the purpose of your program	

Writing Code Basics

Making Variables

Variables store numbers, letters, sentences, sets of data, and everything else you need to run a program. When you need to reference something later, make a variable, assign it a value, and refer to it when you wish to retrieve that value.

- **Declaration and assignment**
  - There is no need for declaration of variables. The type will be implicit with the value when you assign a value to your variable.
- **Type**
  - Data types change based on the value assigned to them. The same variable may have many types over its lifetime.
- **Assigning variables**
  - `x = 1`
    - The variable named *x* has the value 1.
  - `x, y, z = 1, "two", 3.5`
    - You can assign multiple values at once. *x*, *y*, and *z* now have the values 1, two, and 3.5, respectively.
  - `x = y = z = 10`
    - All three of the variables gain the same value, 10.

EX:

```
x = 5
print(x)
x, y = 5.5, 6
print(x + y)
x = y = "five"
print(x + y)
```

- Here, the first print statement will result in "5", the second in "11.5", and the third in "fivefive".
- The variable *x* changes type three times, beginning as an integer, turning into a float, and ending as a string. The variable *y* changes twice, beginning as an integer and changing to a string.

Types

Every variable you make has a data type. Because Python is dynamically typed, the same variable can change type later, but it will only ever have one at a time.

- **Integer (int)**
  - A limitless-digit non-decimal number used for math involving whole numbers
    - `x = 54321`
- **Floating point (float)**
  - A limitless-digit decimal number
    - `x = 123.12345`
- **Complex**
  - A number with one defined and one undefined part (when assigning, you must use `j` or `J` for the undefined part)
    - `x = 5j`

- **String**
  - An array of characters
    - `x = "five"`
- **List**
  - A mutable series of items
    - `x = [1,2,3]`
- **Tuple**
  - An immutable series of items
    - `x = ("one","two","three")`
- **Boolean (bool)**
  - A true or false value
    - `x = True`
- **Dictionary (dict)**
  - A list of items and their identifying keys
    - `x = {1:"One",2:"Two",3:"Three"}`
- **Changing types**
  - Sometimes you need to manually change the type of a variable (e.g., when you need to add an integer as part of a concatenated string). Do this by listing the desired type and enclosing the variable you wish to convert in parentheses.
    - `x = 5`

```
print("The number is " + str(x))
```

Note: This prevents an error.

Console

Many programs require user input. It is equally important to present information directly to the user. An easy way to do this is to use console input and output statements.

- **Outputting text**
  - Turn an object into a string of text and make that text appear to the user with a print statement. If you send more than one object at once, make sure they are of the same type.
    - `print(x)`
- **Taking user input**
  - Allow the user to enter data for your program with an input statement. You can prompt them with a message that suggests what kind of data they are entering.
    - `input("Write a message here: ")`

Error Handling

An invaluable part of any program is the series of data validation and error handling set up to catch unexpected glitches that might sneak through. These safeguards are there to protect the integrity of operations and ensure that nothing is spoiled because of bad data.

- `try:`
  - Everything within the scope of the *try* statement is subject to error handling with an accompanying **except** statement.
- `except:`
  - Everything within the scope of the *except* statement is run when an error is encountered in the **try** statement. If a particular error is specified after the except statement, only that particular error triggers the corresponding except code.
- `else:`
  - Code within the scope of the *else* statement runs if there are no errors found in the **try** statement.
- `finally:`
  - Code within the scope of the *finally* statement executes regardless of whether or not errors were raised while executing code in the **try** statement.
- `raise:`
  - Using *raise* specifies an error that you may manually call or re-raise the last error found if no arguments are passed.
  - `raise(NameError)`

EX:

```
try:
    x = "five"
    y = x + 5
except TypeError:
     print("You must use math types to do math!")
except:
    print("Another error")
else:
    print("The math worked!")
finally:
    print("This code always runs.")
```

In this code, we enclose some simple math in a try statement, but we made an error when deciding on the data to do the math with. Our program would normally crash, but because we are checking for errors, the `except TypeError:` is run instead. If our math had completed successfully, the `else:` code would have run. The `finally:` code will always run.

Saving & Loading Files

Python has utilities for creating, reading from, and writing to files. Use them when you need to get data for your program or log the results produced by your program for future reference.

- `open(filename, mode)`
  - Opens a file that can then be read or updated
  - Takes two arguments, one that designates the name of the file as a string and the other that specifies what you wish to do with the file (`r` = read, `w` = write, `a` = append, and `r+` = read or write)
- `file.read(size)`
  - Reads a specified amount of the file
  - Reads all of the file if size is unspecified
- `file.readline()`
  - Reads from the file until a newline (`\n`) character is encountered
    - The resulting string keeps the newline (`\n`) character at the end.
- `file.write()`
  - Edits the file by adding the contents of a string to the end of it
  - Returns the total number of characters that were added
- `file.close()`
  - Closes the specified file, freeing up space and allowing it to be read again

EX:

```
file = open("essay.txt","r+")
title = file.readline()
print(title)

body = file.read()
print(body)

file.write("\nThe End!")
file.close()

file = open("essay.txt","r")
print(file.read())
file.close()
```

Here we open a file named `"essay.txt"` with the capability to both read from and write to it. We use `file.readline()` to get the title (the first line) and `file.read()` to get the remaining lines, printing each in turn. We use `file.write()` to add another line at the end of the file, and then close the file. To see what the entire file looks like, we re-open it, print the entirety of it, and then close it.

Coding Structures

List Operations (`list`, `tuple` & `dict`)

List-based data types are great for holding pieces of related data. Especially as programs get larger, using the list-based data types becomes crucial for managing many elements. There are a lot of list-based operations, and many of them function for all three list types.

- `x = []`
  - Creates a new list named *x*
- `x.append`
  - Adds to the end of list *x*
- `x.pop`
  - Takes away from the end of list *x*
- `x.remove(y)`
  - Takes away the element *y* from list *x*
- `x.insert(y, z)`
  - Adds the element *z* at the position *y* of list *x*
- `del x[y]`
  - Removes the element at the position *y* from list *x*
- `x.index(y)`
  - Gets the position of the element *y* in the list *x*
- `x.extend(y)`
  - The list *x* now contains the list *x* and the list *y* concatenated
- `x.clear()`
  - Gets rid of all the items in the array
- `x[y]`
  - Gets the element *y* at the position in the list *x*
- `x[-1]`
  - Gets the element in the last position in list *x*
- `x[y:z]`
  - Gets elements in list *x* from (including) *y* position to (excluding) *z* position
- `x[y:]`
  - Gets elements in list *x* from (including) *y* position to the end
- `x[:z]`
  - Gets elements in list *x* from the beginning until (excluding) *z* position
- `x[::y]`
  - Gets elements at every *y* position in list *x*
- `x[::-1]`
  - Gets elements in list *x*, reversed
- `y = x[:]`
  - Makes *y* into a copy of list *x*
- `y in x`
  - Checks if *y* is in *x* (returns true or false)
- `len(x)`
  - Returns the number of items in list *x*

You can see how the list operations change or measure the values contained in the list.

List Operations

List operation	Process	Result
`x = [1,2,3,4,5,6,7,8,9,10]`	The list is created with numbers [1–10].	`[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`
`x.append(11)`	[11] is added to the end of the list.	`[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]`
`x.pop()`	The element at the end [11] is removed.	`[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`
`x.remove(3)`	The element [3] is removed.	`[1, 2, 4, 5, 6, 7, 8, 9, 10]`
`x.insert(0,0)`	Element [0] is added at the first (0) position.	`[0, 1, 2, 4, 5, 6, 7, 8, 9, 10]`
`del x[6]`	Element at position (6) is removed.	`[0, 1, 2, 4, 5, 6, 8, 9, 10]`
`print(x.index(10))`	The position (8) of element [10] is returned.	`8`
`print(x[4])`	Element [5] at position (4) is returned.	`5`
`print(x[-1])`	Element [10] at the final position is returned.	`10`
`print(x[2:6])`	Elements that begin at position (2) and end at position (6) are returned.	`[2, 4, 5, 6]`
`print(x[4:])`	Elements that begin at position (4) and end at the end of the list are returned.	`[5, 6, 8, 9, 10]`
`print(x[:4])`	Elements that begin at the start of the list and end at position (4) are returned.	`[0, 1, 2, 4]`
`print(x[::3])`	Elements at every third position are returned.	`[0, 4, 8]`
`print(x[::-1])`	The list is returned, in reverse order.	`[10, 9, 8, 6, 5, 4, 2, 1, 0]`
`print(5 in x)`	The result of checking the list for element [5] is returned.	`True`
`print(len(x))`	The length of the list is returned.	`9`

Math Operators (`int`, `float` & `complex`)

Almost all programs have elements that require performing calculations. To do this, you can use numbers in conjunction with math operators, which are characters that hold meaning when used to manipulate these number values.

- **+ add**
  - `print(5 + 5)`
    - Returns: `10`
- **- subtract**
  - `print(10 - 5)`
    - Returns: `5`
- **/ divide**
  - `print(100 / 20)`
    - Returns: `5`
- *** multiply**
  - `print(2.5 * 2)`
    - Returns: `5.0`

- **// divide (floor)**
  - `print(11 // 2)`
    - Returns: `5.0`
- **% remainder**
  - `print(17 % 6)`
    - Returns: `5`
- **** exponent**
  - `print(2 ** 2)`
    - Returns: `4`
- **() perform enclosed operations first**
  - `print((10-9)*5)`
    - Returns: `5`
- **abs() find absolute value**
  - `print(abs(5 - 10))`
    - Returns: `5`

Strings

Strings are text values without any inherent meaning, mathematical or logical.

Creating String Literals

- A string literal is an explicitly typed string, which means that it contains a series of set characters typed in a specific order.
- You may use either single or double quotation marks around a string literal.
- `x = "This is a string"`

Quotations within Strings

- If you need to put quotation marks inside a string, you may use the opposite kind of quotation marks.
- `x = 'Quothe the Raven, "Nevermore."'`

Adding Strings Together

- Using the + sign will concatenate strings (i.e., put them together).
- `x = "This string" + "That string"`

Getting a Character from a String

- Strings are essentially lists of characters. You can get a particular character from a string by enclosing its position in brackets [].
- `x = "String"`
- `print(x[0])`
  - Returns: `S`

Getting the Length of a String

- Similar to other lists, you can get the length of a string by using `len()`.
- `print(len("five"))`
  - Returns: `4`

Statements

Statements are meaningful commands in Python that initiate some specific operation. These are crucial tools used to iterate through data and branch the program into different scenarios based on input values.

Boolean Statements

- `if x:`
  - An `if` statement initiates a chain of logic that breaks the code execution into two possible paths. If the statement is true, the code executes; otherwise, the code is skipped.
- `elif y:`
  - After an `if` statement, if you wish to check for additional conditions, you may use an `elif` statement, which will only run when the original `if` statement is false.
- `else:`
  - Placed after an `if` or `elif` statement, the else statement will run only if all preceding statements are false.

EX:

```
x = input("Please list a fruit: ")
if x == "apple":
    print("Apples are good!")
elif x == "banana":
    print("Bananas are delicious!")
else:
    print("I've never had a fruit like that.")
```

The `if` statement checks the user's input to see if it is the string "apple". If it is, the program outputs a response; otherwise, it moves on to check if the input is "banana", outputting a response if so. If the input is neither apple nor banana, the `else` statement runs, providing a response for all other cases.

For Statements

- `for x in y:`
  - This statement iterates through every element in *y*, with each element being referred to as *x* for the duration of the code block.

```
x = ["apples","bananas","cherries"]
for fruit in x:
    print(fruit + " are delicious!")
```

  Note: This prints a statement for each element in *x*.
- `for x in range(y):`
  - The statement iterates through a range of numbers ending at (excluding) *y*.

```
for x in range(5):
    print(x)
```

  Note: This prints numbers 0 through 4.
- `for x in range(y, z):`
  - This statement iterates through a range of numbers starting with (including) *y* and ending with (excluding) *z*.

```
for x in range(5,10):
    print(x)
```

  Note: This prints numbers 5 through 9.
- `for x in range(y, z, a):`
  - This statement iterates through a range of numbers starting with (including) *y* and ending with (excluding) *z* at intervals of *a*.

```
for x in range(0,10,2):
    print(x)
```

  Note: This prints numbers 0, 2, 4, 6, and 8.

While Statements

- `while x:`
  - This while statement repeatedly executes code in its scope as long as condition *x* is true.

EX:

```
x = 0

while x < 10:
    print(x)
    x += 1
```

With every loop through this statement, *x* will be printed, and the value of *x* will increase by 1. This will go on until *x* is no longer less than 10.

Functions

Functions are an important part of any even remotely complicated program. They are the tools used to break down operations into smaller, reusable chunks that can be called on in many situations to fill different needs.

- **Function overview**
  - Functions should be versatile, reusable, and independent.
  - When a function's scope ends, any variables created inside of it and not returned will be inaccessible.
- **Creating a function**
  - `def function_name():`
- **Calling a function**
  - `function_name()`
- **Functions with parameters**
  - These functions require some kind of data to be sent to them in order to run. Incorrect or missing data will result in an error.
  - `def function_name(parameters):`
- **Default parameters**
  - Functions with default parameters are functions that can be run whether or not data is sent. If no data is sent, the defaults for the parameters will be used.
  - `def function_name(parameter="default data"):`
- **Return statement**
  - This will cause the function to end and can be accompanied by a value that will be sent back.
  - When using a return statement, you will want to set a variable that will receive the returned value from the function.

EX:

```
def findGPA (percentgrade):
    return percentgrade*4


def studentgrade (name,grade="unknown"):
    print(name + " has a GPA of " + str(grade))

gpa = findGPA(.86)
studentgrade("Frank",gpa)
studentgrade("Ellen")
```

Two functions are defined, `findGPA` and `studentgrade`. One takes a single parameter (`percentgrade`) and the other takes two parameters (`name` and `grade`), with `grade` being optional and having a default value of `"unknown"`. The code runs, first defining the variable `gpa` by running the first function, and then using the second function to print two students' grades, one where we know the GPA and one where we do not.

Dictionaries

Dictionaries are special list types that contain key value pairs. They are two-dimensional arrays with unique identifier keys attached to values.

Keys must be of immutable data types and contain only unique values (i.e., never repeat).

- `dictionary = {"Key1":"Value1","Key2":"Value2"}`
  - Written using braces and pairs of data separated by colons
- `dict.fromkeys(list,value)`
  - Creates a new dictionary using keys from the provided list and setting all associated values to the provided value
- `dict.get(key,default)`
  - Gets the value associated with a key in the referenced dictionary
    - **Note:** If the key is not found, it returns the default value.
- `dict.items()`
  - Gets a list of all keys in the dictionary and the values associated with them
- `dict.keys()`
  - Gets a list of all keys in the dictionary
- `dict.values()`
  - Gets a list of all values in the dictionary
- `dict.update(dictionary)`
  - Adds the contents of one dictionary to another

EX (dictionaries):

```
USCapitals = {"Alabama":"Montgomery","Alaska":"Juneau","Arizona":"Pheonix","Arkansas":"Little Rock"}
print("The capital of Alabama is " + USCapitals.get("Alabama","Unknown"))
print("The capital of Colorado is " + USCapitals.get("Colorado","Unknown"))

USCapitalsContinued = {"California":"Sacramento","Colorado":"Denver","Connecticut":"Hartford"}
USCapitals.update(USCapitalsContinued)
print("The capital of Colorado is " + USCapitals.get("Colorado","Unknown"))
print(USCapitals.items())
```

- Here we make a dictionary called `USCapitals` that contains the start of a list of all the states and their capital cities. We print some information about Alabama by getting the capital city associated with the state. We try the same thing for Colorado, but there is no key for it yet.
- We make a second dictionary containing more state capitals and use `dict.update()` to add it to the first dictionary. We can find the relevant information and see all of the information at once by printing every item contained in the dictionary.

Using Structures

String Formatting

Often, you will want to output different strings, with the results being dependent on values in your program. Formatting strings allows you flexibility when doing this and gives you options for variable content inside the string itself.

- **Using `string.format`**
  - This method of string formatting comes after the declaration of a string and replaces certain placeholder elements declared in the string with alternate data.
  - The three placeholder elements are empty braces {}, braces with numbers {0}, and braces with words {word}.

EX:

```
print("This {} into the {}.".format("inputs","sentence"))
```

Returns: `This inputs into the sentence.`

```
print("{0} think, therefore {0} {1}.".format("I","am"))
```

Returns: `I think, therefore I am.`

```
print("The {object} is {property}.".format(object="feather",
property="light"))
```

Returns: `The feather is light.`

- **Using other formatting options**
  - There are two other methods of formatting strings, one that uses an `f` before the string to replace parts with preexisting variables, and one that works like empty braces but uses `%s` and is compatible with older versions of Python.

EX:

```
animal = "panther"
habitat = "jungle"
print(f"The first letter of {animal} is {animal[0]}, and {animal}s
live in the {habitat}.")
```

Returns: `The first letter of panther is p, and panthers live in the jungle.`

```
print("%s works in %s versions." %("This","old"))
```

Returns: `This works in old versions.`

String Methods

- **Slicing**
  - `String[x:y]`
  - Returns a string encompassing the characters from position *x* to position *y*
- **Reversing**
  - `String[::-1]`
  - Returns the same string, but in reverse
- **Splitting**
  - `String.split(character)`
  - Splits a string into a list of strings using the specified character to determine where to split it
- **Joining**
  - `Character.join(list)`
  - Takes a list of strings and puts them together in order, using the specified character between each one
- **Enumerating**
  - `Enumerate(string)`
  - Turns the string into matched number-character tuples

EX:

```
Truth = "This is a great guide with lots of valuable information!"
ShortTruth = Truth[0:21]
print(ShortTruth)

TruthWords = ShortTruth.split(" ")
print(TruthWords)

TruthParagraphs = "\n".join(TruthWords)
print(TruthParagraphs)

ReversedTruth = TruthParagraphs[::-1]
print(ReversedTruth)

EnumTruth = enumerate(ReversedTruth)

FinalWords = ""
for pair in EnumTruth:
    FinalWords = pair[1] + FinalWords
print(FinalWords)
```

Here we alter the original string, `Truth`. First, we slice it, returning only the first 22 characters, then we split it into a list of words by separating the sentence using each space. Then we join it back together using a newline character between words, and then we reverse the sentence, turning it into garbled letters. We take these letters and turn them into enumerated pairs, create a placeholder for our final word, and loop through those enumerated pairs, placing the second element of each one (the letter) at the beginning of the word, effectively returning a sentence that has been reversed again.

Escape Sequences

Escape sequences are special sets of characters that change the meaning of the characters that follow them. In Python, the \ character is used both to give meaning to characters (e.g., \n) and to remove meaning from characters (e.g., when placed before a ").

- **\ newline**
  - A \ followed by the enter key ignores the creation of the new line, allowing multiline code and text.
  - `print("Sometimes you need to write your code\`
    `across a few different lines without breaking the line of code syntactically.")`
- **Printing special characters literally**
  - Placing a \ before the characters \, ', and " allows them to be printed literally.
  - `print("We print \"sentences\" using the \\ character sometimes.")`
    - **Returns:** `We print "sentences" using the \ character sometimes.`
- **\n**
  - This character combination starts a new text line.
  - `print("In a short haiku, \nthe lines must be broken up \ninto only three.")`
    - **Returns:**

```
In a short haiku,
the lines must be broken up
into only three.
```

- **\t**
  - This character combination places a tabular indentation.
  - `print("\tWe can indent paragraphs of text.")`
    - **Returns:** `We can indent paragraphs of text.`

Bool Characters

These characters are used to perform operations related to true and false statements.

- `True (1)`
  - This value means true, 1, or any non-zero value, depending on context.
- `False (0)`
  - This value means false, 0, or any empty array, depending on context.
- `not`
  - This checks for the opposite of the following Boolean value.
- `and`
  - When you use this operator, all statements must be true or the whole statement is false.
- `or`
  - When you use this operator, if any statements are true, the whole statement is true.
- `==`
  - This comparison checks for equivalent value.
- `!=`
  - This comparison checks for unequal values.
- `>`
  - This comparison checks if the first value is greater than the second.
- `<`
  - This comparison checks if the first value is less than the second.
- `>=`
  - This comparison checks if the first value is greater than or equal to the second.
- `<=`
  - This comparison checks if the first value is less than or equal to the second.
- `x < y and y < z`
  - This means the same thing as $x < y < z$. This checks to ensure that the value *y* lies between the other two values.
- `is`
  - This checks to see if a given variable references the same object as another variable.

Writing Boolean Statements

There are a few ways to execute code based on the value of Boolean statements. These statements are used when the program must choose between several code paths and use the Boolean operators and statements that were mentioned earlier.

- **Alternate if-else**
  - `a if x else b`
  - This way of writing an if-else statement executes code *a* if condition *x* is true and executes code *b* in all other cases.
- **Nesting if**
  - You can form advanced chains of logic by nesting one logical statement inside another to test conditions only if other conditions have proven true.

EX:

```
print("Enter your age and weight to find out your lucky number.")
age = int(input("Please enter your age(in years): "))
weight = int(input("Please enter your weight(in pounds): "))
if age < 20:
    if weight > 80:
        print("Seven")
    else:
        print("Twenty-four.")
elif age < 60:
    print("Four") if weight > 80 else print("Fifteen")
else:
    print("Ninety-nine")
```

Two different integer values, stored in the variables `age` and `weight`, are gathered from the user, and they will be used to determine which number is printed. Using Boolean statements, we check the first value, determining with an if-elif-else statement whether it is less than 20, less than 60, or larger than 60. Within those logic trees, we then nest additional logic to measure the weight, printing different numbers depending on whether or not it is larger than 80.

Recursion & Iteration

There are two great ways to automate operations so you can make many processes with a minimal amount of code. When functions call themselves, it is called recursion. This is generally done when you can break a large operation into many instances of the same operation. When you loop through all the objects in a set of objects, it is called iteration. You do this when you want to perform one block of code many times with different objects.

- **Recursion**
  - This occurs when a function calls itself inside itself.
  - You must include some sort of condition which will change sufficiently to end the recursion, or the function will never end.
- **Iteration**
  - Use for or while loops and any iterator object (e.g., arrays or structs).
  - Iterative objects contain a series of items.
- **Break**
  - These are used in iteration to stop the current loop and leave the current level of iteration.
- **Continue**
  - These are used to stop the current loop and move on to the next element in the iteration.

EX (recursion):

```
def addchain(number, increment):
    if num <= 0:
        return 0
    else:
        return num + addchain(number-increment, increment)

print(addchain(10,2))
```

The addchain function takes two parameters. One will be the number to start from, and the other will be an increment that counts through the numbers leading up to that number at an interval. The function will add every number starting at the first parameter and ending at zero, at increments specified by the second parameter. The print function will show us the result of doing so with 10 at increments of 2 and will print 30 as the result. Notice that the function is recursive. It continues calling itself, sending new parameters for the number-increment, ending only when the number sent is less than or equal to zero, at which point all the functions resolve.

EX (iteration):

```
fruits = ["apple","banana","strawberry","cherry","watermelon"]
wantfruit = "strawberry"
for fruit in fruits:
    print(f"Would you like a {fruit}?")
    if fruit == wantfruit:
        print(f"\tYes, I'll take a {fruit}, thank you!")
        break
    else:
        print(f"\tNo, thank you, that's not what I want.")
```

One variable represents an array of fruits and one represents a desired fruit. A `for` statement iterates through the array, printing a question and checking each fruit to determine whether it matches the desired fruit. If it does not, we print a response refusing the fruit. If it does, we print a response accepting it. When the fruit is accepted, we use a `break` statement to stop the current iteration, because once we have selected our fruit, we do not need to check through the remaining fruits.

Classes

Classes are collections of related functions and variables that act as templates for arranging different sets of data.

- **Creating a class**
  - Classes need two parts, the **declaration**, which is similar to the declaration of a function, and the **initialization**, which is a function within the class responsible for creating new members of it.
- **Filling a class**
  - Classes have associated functions and variables that are attached to the class and identified as properties of it.

EX (classes):

```
class fruit:
    def __init__(self, name, color, shape, texture, insidec, insidet):
        self.name = name
        self.color = color
        self.shape = shape
        self.texture = texture
        self.insidecolor = insidec
        self.insidetexture = insidet
    def aboutme(self):
        print(f"I am a {self.name}, I am shaped like a {self.shape}, and I am {self.color}! If you touch me, I feel {self.texture}.")

    def peel(self):
        self.color = self.insidecolor
        self.texture = self.insidetexture

mybanana = fruit("banana","yellow","crescent","smooth","white","sticky")
myapple = fruit("apple","red","sphere","smooth","white","porous")
myorange = fruit("orange","orange","sphere","rough","yellow","porous")

mybanana.aboutme()
mybanana.peel()
mybanana.aboutme()

print(f"My apple is {myapple.color} on the outside and {myapple.insidecolor} on the inside.")
print(f"My orange feels {myorange.texture}, and my apple feels {myapple.texture}.")

myorange.color = "blue"
print(f"I painted my orange, and the color is now {myorange.color}.")
```

- There is a class named fruit with many related properties and functions. Three instances of that class are created, one banana, one apple, and one orange. When they are created, we define the necessary properties of the fruit so that we may refer to them later.
- When we wish to call related functions, we do so by using the name of one of the class instances we created, a period, and the name of the function of property we wish to access. We call the `aboutme` function to print a statement that contains information about the instance of `mybanana`. When we then call the `peel` function and use the `aboutme` function again, we can see how the properties have changed to reflect the new information.
- We can access properties of all of the instances of this class we have created, and we can see this in the `print` statements. They access information about the objects `myapple` and `myorange` in order to display the relevant information about them.
- We may access these properties directly and alter them, such as in this case, where we set the color of the `myorange` object to be blue. Once we do so, when we access that property, the new value will be returned.

Coding Concepts

Inheritance

Inheritance is the process by which objects gain the properties of other objects. In Python, inheritance happens with classes. The first class, called the parent, has properties that the second class, called the child, gains by default in addition to its own set of properties.

- **Parent class**
  - This is a class of the type object, and it contains base information that all of its children inherit.
- **Child class**
  - This is a class of the type created by the parent class and expands upon the information of that class.
- **Inheritance and identity**
  - Classes that inherit from a parent class are considered members of that class for the purpose of identification.
- **Overriding**
  - Children can overwrite the inherited properties of their parent, changing the functionality.
- **Inherit from multiple parents**
  - Classes can inherit from many parents, gaining the properties of all of them.

EX:

```
class mammal:
    circulation = "is warm blooded"
    skeleton = "is a vertebrate"
    skin = "has hair"
    parent = "produces milk"
class bird:
    circulation = "is warm blooded"
    skeleton = "is a vertebrate"
    skin = "has feathers"
    parent = "lays eggs"
class human(mammal):
    def __init__(self,myname):
        self.myname = myname
```

```
class crow(bird):
    def __init__(self,myname):
        self.myname = myname

class birdman(bird,human):
    skin = human.skin + " and " + bird.skin
    parent = human.parent + " and " + bird.parent

frank = human("Frank")
print(frank.myname + " " + frank.circulation)
print(frank.myname + " " + frank.skin)

alice = birdman("Alice")
print(alice.myname + " " + alice.parent)
print(alice.myname + " " + alice.skin)

greg = crow("Greg")
queue = [frank,alice,greg]
for animal in queue:
    if isinstance(animal,mammal):
        print(animal.myname + ", who " + animal.skin + ", is standing in line.")
    elif isinstance(animal,bird):
        print(animal.myname + ", who " + animal.skin + ", is flapping around.")
```

- Here we set up the `mammal` and `bird` classes, which serve as parent classes. They contain properties that any derived class inherits. We then create three more classes, `human` (a member of the mammal class), `crow` (a member of the bird class), and `birdman` (a member of the bird and human classes, and therefore a member of the mammal class).
- With classes created, we create `frank`, an instance of the human class, passing an argument that represents frank's name. We can print some information about frank that comes from its human status (`myname`) and its mammal properties (circulation and skin).
- The same is true of the next instance we create, `alice`, of the birdman class. The birdman class inherits properties from both the bird and the human classes. Normally, the properties from the bird class and the mammal class would conflict, so alice would have only those properties from the first class it inherits from. In this case, the birdman class contains properties that overwrite the properties of its parent classes (skin and parent).
- We make an instance of the crow class (`greg`) and create an array containing the three class instances. In the following for loop, each element in that array is examined. Using the `isinstance()` method, we check whether that element is a mammal and print that it is standing if it is. If it is not a mammal, we check if it is a bird and print that it is flapping around if it is. In this case, even though the array elements are the classes human, birdman, and crow, they are also considered members of their parent classes. We see that frank and alice stand in line while greg flaps around.

Generators

Generators are special functions that make iteration and memory management easier. Unlike other functions, they retain information relating to their contents even after returning a value, allowing them to be resumed later, at which point they start from where they left off.

- `yield x`
  - Makes a function become a generator by returning a value while maintaining its position in the function
- `next()`
  - Used to navigate to the next value yielded in an instanced generator

EX:

```
def generatesequence ():
    number = 0
    add = 1
    while True:
        yield number
        number += add
        add += number

sequence = generatesequence()
print(next(sequence))
```

- Here we create a simple function that generates a sequence of numbers. We use the `yield` keyword to indicate that the starting number will be sent back when the sequence is started. After that, the number will be incremented at a constant rate and sent back during each continuation of the function.
- We can see this when we create a sequence based on the generator function and print the first element using `next(sequence)`. Every time we repeat this, it creates another element in the sequence.

Polymorphism

Polymorphism is the utilization of the same function for many purposes. This can mean that the function takes different arguments or works differently for different objects.

- Polymorphism with inheritance
  - A child class can take methods from its parent class and override parts of them.
- **Polymorphism with classes**
  - Two different classes can use methods of the same name to produce different results.
- **Polymorphism with types**
  - Multiple methods can be created that take different types as arguments, allowing them to be called in many situations.

EX:

```
class dog():
    def __init__(self,name):
        self.name = name
    def speak(self):
        return "Arf! Arf!"

class chihuahua (dog):
    def speak(self):
        return "Yip! Yip!"

class cat ():
    def __init__(self,name):
        self.name = name
    def speak(self):
        return "Meow! Meow!"

def teach (animal):
    print(animal.name + " makes the noise: " + animal.speak())

balto = dog("Balto")
spike = chihuahua("Spike")
sally = cat("Sally")

print(balto.speak())
print(spike.speak())
print(sally.speak())

teach(spike)
teach(balto)
teach(sally)
```

- Here we create the `dog`, `chihuahua`, and `cat` classes. All three classes have the `speak` method, and each one does something different. Through polymorphism, we can use the same syntax to call the method from instances of all three objects. For the chihuahua class, we see polymorphism by overriding the method of the parent class; for the cat class we see polymorphism by having the same method in different classes.
- We also can see polymorphism at work in the `teach` function, which takes one argument. The function can take any object which has the name and speak attributes, as we can see when we call the method using all three animal instances as arguments.

Lambda Expressions

These are functions that can be created on the spot without even being named. They have one expression and return the result of it.

- `lambda values : expression`
  - Takes the values given and uses them to evaluate the expression, returning the result
- **Passing the lambda expression**
  - You may pass the lambda expression to a variable, which will allow you to access the function by that variable name at a later time.

EX:

```
def exponent(e):
    return lambda x : x ** e

square = exponent(2)
cube = exponent(3)
print(square(10))
print(cube(10))
```

Here we create a function that returns a `lambda` function. That function creates a new function that takes a value and multiplies it exponentially by the number sent. This lets us create any number of exponent-based functions by inputting different values. This is apparent when you print the results of the newly created `square` and `cube` functions, which return 100 and 1,000, respectively.

$7.95 **Author:** Berajah Jayne

ISBN-13: 978-142324188-1
ISBN-10: 142324188-6
50795