Python Basic Information

A general purpose programming language
- can be used for many different things

A high level programming language
- abstracts away from machine code
- needs to run through a python interpreter

Here are the two repos that will be used as a reference in this cheat sheet: **https://github.com/iamshaunjp/python-3-playlist**
https://github.com/Richard-Burd/python-3-sandbox

```
user@ubuntu ~/any/path/here $ python -V
// => This is the default version of Python installed; make sure you use a capital 'V', not 'v'

user@ubuntu ~/any/path/here $ python
// => This starts the python shell where you can type in Python code directly (press "Ctrl+Z" to exit the shell)

user@ubuntu ~/any/path/here $ python3
=> This starts the Python version 3 shell and since Ubuntu uses Python 2.7 you have to have 2.7 installed

user@ubuntu ~/any/path/here $ python3 myPythonScript.py
=> This will run a Python file called "myPythonScript.py" through the Python version 3 interpreter
```

This is the shell your python code is being passed through in order to execute

This is the actual code being executed

Everything in Python is cnsidered to be an object, and objects have attributes and functions; when we talk about these functions with respect to these objects, we call them methods…thus, objects can have attributes and methods.

Numbers

Python has two types of numbers; integers and floats. You can find the data type by using the built in **type()** method

```
type(500)
# <class 'int'>

type(5.1)
# <class 'float'>

type("Hey!")
# <class 'str'>
```

Integers
- are whole numbers

Floats (or floating point numbers)
- anything with a decemal in them
- needs to run through a python interpreter
- is always the result of division

Pythonistas call this an 'argument'

```
5 / 5
# 1.0
```

… unless you use the doulbe forward slash

```
5 // 5
# 1

10 % 3
# 1
```

division remainder or "modulus"

//

BIDMAS stacking order

+=

```
(5 + 3) * 2 + 2**3 - 6
# 18
```

Brackets
Indicies
Multiplication
Addition
Subtraction

```
age = 5
age = age + 5
# age => 10

age = 5
age += 5
# age => 10

age *= 5
age /= 5
age -= 5
```

This is a shorter way of expressing what is to the left; these also exist

"Strings" & [Lists]

```
"I'm doing good!"
'I\'m doing good!'
```

You can escape a character with the backslash

```
greet = "Hello!"
greet[0]
#=> 'H'

greet[-1]
#=> '!'

greet[0:4]
#=> 'Hell'

greet[0:-2]
#=> 'Hell'
```

We start with **0** when counting string elements from the front and **-1** when counting string elements from the back

slicing

We can start with 0 and count up to (but not including) the 4 when grabbing a sequence of elements in a string; we can also count **from the back**

.upper()

```
greet = "Hello!"
noun = "Cat"
greet + noun
#=> 'HelloCat'

noun * 3
#=> 'CatCatCat'

noun.upper()
#=> 'CAT'

noun.lower()
#=> 'cat'

len(noun)
#=> 3
```

add (concatenate) & multiply strings

Python has special methods for strings

```
cheeses = "Brie, Chedder, Stilton"
cheeses.split(',')
#=> ['Brie', 'Chedder', 'Stilton']
```

you can split a string into a list [in python, arrays are called "lists"] at a specified character

.split()

Length of a string or list can be found with **len()**

.append()

```
list1 = ['Bob', 'Tom', 'Ron']
list1[1]
#=> 'Tom'

list2 = list1 + ["Cat"]
#=> ['Bob', 'Tom', 'Ron', 'Cat']

list2.append('Dan')
#=> ['Bob', 'Tom', 'Ron', 'Cat', 'Dan']

list2.pop()
#=> 'Dan'
list2 = ['Bob', 'Tom', 'Ron', 'Cat']

list2.remove('Tom')
#=> ['Bob', 'Ron', 'Cat']

del(list2[0])
#=> ['Ron', 'Cat']

nest = [[1, 2], [3, 4], [5, 6]]

nest[2][1]
#=> 6
```

find the index of a list item

concatenate elements onto a list

add at the end of a list

.pop()

remove the last element in a list
.pop() removes last element from object
.pop() returns the last element, not the object

WARNING, this only removes the 1st instance of 'Tom'

remove a selected index from a list

find nested list element

del([])

[List] Comprehension

Given a list of numbers we want to double; there are two different methods for doing this shown below

Standard Method

```
numbers = [5, 10, 50, 100, 1000]
double_numbers = []

for number in numbers:
  double_numbers.append(number*2)

print(double_numbers)
#=> [10, 20, 100, 200, 2000]
```

Comprehension Method

```
numbers = [5, 10, 50, 100, 1000]
double_numbers = [number*2 for number in numbers]

print(double_numbers)
#=> [10, 20, 100, 200, 2000]
```

same output

Another example below squares all even numbers from 1 to 10

Standard Method

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

squared_even_numbers = []

for num in nums:
  if(num**2) % 2 == 0:
    squared_even_numbers.append(num**2)

print(squared_even_numbers)
#=> [4, 16, 36, 64, 100]
```

Comprehension Method

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

squared_even_numbers = [
  num**2 for num in nums if(num**2) % 2 == 0
]

print(squared_even_numbers)
#=> [4, 16, 36, 64, 100]
```

same output

The kind of comprehension method can be used (mutatis mutandis) on dictionaries as well

[List] & (Tuple) Manipulation

all code shown in this section is available here:
github.com/Richard-Burd/python-3-sandbox/timfiles/list_&_tuple_manipulation.py

Tuples are used for coordinates, colors, rectangles, & other mathy stuff; they are similiar to lists

```
my_list = ["a", "b", "c", "d", "e", "f", "g", "h"]

my_tuple = ("a", "b", "c", 4, "e", "f", "g", "h")

print(my_tuple[slice(3, 5)])
#=> (4, 'e')

print(my_list[slice(3, 5)])
#=> ['d', 'e']

my_list.append("i")

my_list[-1] = "z"

print(my_list)
#=> ['a', 'b', 3, 'd', 'e', 'f', 'g', 'h', 'z']
```

Lists & tuples can both contain mixed data types and nested elements; the **slice()** method can be used on both of them,

Tuples are immutable and lists are mutable; in example:
tuples cannot be appended but lists can be appended

Decorators

Decorators extend the behavior of a function without modifying the function itself; they are use extensively in web frameworks like Django.

```
def cough_discount(function):
  def function_wrapper():
    print('*cough*')
    function()
    print('*couuugh*')

  return function_wrapper

@cough_discount
def question():
  print('can you give me a discount for that!?')

question()
#=> *cough*
#=> can you give me a discount for that!?
#=> *couuugh*
```

Decorators are basically wrapper functions; they are defined with an **"@"** symbol & the decorator function name must **match** the decorator istelf

This code runs before the function

This code runs after the function

Conditionals

User inputs are always accepted as strings (e.g. "2") unless they are type-casted with **int()** to become integers

if : elif : else :

```
age = int(input('Enter your age:'))

if age < 10:
  print('you are under 10')
elif age >= 10 and age <= 20:
  print('You\'re a teenager ')
elif age == 33:
  print("You're middle aged")
else:
  print("That is old")
```

Python uses colons and needs the following line to be indented

Python uses the **and** keyword instead of && like in Ruby or JavaScript

if

```
ninjas = ['ryu', 'shu', 'snu', 'moe', 'bob', 'dua']

for ninja in ninjas[1:3]:
  print(f'we have: {ninja}')
# => we have: shu
# => we have: snu

for ninja in ninjas:
  if ninja == 'moe':
    print(f'{ninja} - is the last ninja')
    break
  else:
    print(ninja)
# => ryu
# => shu
# => snu
# => moe - is the last ninja
```

starts with index 1 and goes up to, **but not including,** index 3

for in :

after moe, break out of the loop and ignore the other elements

break

```
age = 25
num = 0

while num < age:
  if num == 4:
    print("That's number 4! OK, let's continue on...")
    num += 1
    continue
  if num % 2 == 0:
    print(num)
  num += 1
```

this will continue on with the iteration after doing the stuff above

this will give us only even numbers

while

without these specific indentations, the Python code will not work

continue

File Importing

To import relative from the top level directory, use **dot notation** like this

.python-3-sandbox/**test.py**

```
from lessons.space.planet import Planet

print(Planet)
#=> <class 'lessons.space.planet.Planet'>
```

To import from beyond the top level package, you will need **this block of code**

.python-3-sandbox/files/**test.py**

```
import sys
sys.path.append("..")

from lessons.space.planet import Planet

print(Planet)
#=> <class 'lessons.space.planet.Planet'>
```

Reference: **github.com/Richard-Burd/python-3-sandbox**

Dictionaries

Python dictionaries are similiar to Ruby hashes or JavaScript objects

```
people_ages = {'ron':12, 'bob':5, 'tom':36, 'dan':41, 'cat':39, 'alf':73}

print('ron' in people_ages)
#=> True

print(people_ages['ron'])
#=> 12
```

see if key exists in the dictionary with **in**

find the value of a specified key

in

grabbing the keys in a dictionary will give a value like this

```
print(people_ages.keys())
#=> dict_keys(['ron', 'bob', 'tom', 'dan', 'cat', 'alf'])

print(list(people_ages.keys()))
#=> ['ron', 'bob', 'tom', 'dan', 'cat', 'alf']

print(list(people_ages.values()))
#=> [12, 5, 36, 41, 39, 73]
```

list

either the dictionary **keys** or **values** can be returned as a list (array) data type with the **list** keyword

```
my_names = list(people_ages.keys())
print("there's ", my_names.count('ron'), "ron in the list!")
#=> there's 1 ron in the list!
```

get the number of times a given key is found in the dictionary with the **count** method

```
person = dict(name='shaun', age=27, height=6)

print(person)
#=> {'name': 'shaun', 'age': 27, 'height': 6}
```

This is an alternative syntax for declaring a dictionary, when called, it uses **":"** but uses "=" in the declaration

```
ninja_belts = {}

while True:
  ninja_name = input('enter a ninja name:')
  ninja_belt = input('enter a belt color:')

  ninja_belts[ninja_name] = ninja_belt

  another = input('add another? (y/n)')
  if another == 'y':
    continue
  else:
    break

print(ninja_belts)

def ninja_intro(dictionary):
  for key, val in dictionary.items():
    print(f'I gotta key of: {key} and a value of: {val} !')

ninja_intro(ninja_belts)
```

This takes the user input from the console and sends it to the dictionary

continue

this keeps the code execution in the while loop when the user enters in "y"

items

This function cycles through the dictionary

for…in

This function call starts the program in the console

{Dictionary} Comprehension

Reference: **https://www.datacamp.com/community/tutorials/python-dictionary-comprehension**

Dictionary comprehension is a powerful concept and can be used to substitute for **for** loops and **lambda** functions. However, not all **for** loops can be written as a dictionary comprehension but all dictionary comprehensions can be written with a **for** loop.

.python-3-sandbox/**dictionary_comprehension.py**

Comprehension Method

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

double_dict = {k:v*2 for (k,v) in dict1.items()}

double_dict_if_even = {k:v*2 for (k,v) in dict1.items() if v%2 == 0}

dict1_tripleCond = {k:('even' if v%2==0 else 'odd') for (k,v) in dict1.items()}

print(double_dict)
#=> {'a': 2, 'b': 4, 'c': 6, 'd': 8, 'e': 10}

print(double_dict_if_even)
#=> {'b': 4, 'd': 8}

print(dict1_tripleCond)
#=> {'a': 'odd', 'b': 'even', 'c': 'odd', 'd': 'even', 'e': 'odd'}

nested_dict = {'first':{'a':1}, 'second':{'b':2}}
float_dict = {
  outer_k: {float(inner_v) for (inner_k, inner_v) in outer_v.items()}
  for (outer_k, outer_v) in nested_dict.items()
}
print(float_dict)
#=> {'first': {1.0}, 'second': {2.0}}

people_dictionary = {'member_01':{'age':20, 'name':'Sal'}, 'member_02':{'age':23, 'name':'Tom'}}
people_names = {
  outer_k: {inner_v
    for (inner_k, inner_v) in outer_v.items() if inner_k == 'name' }
    for (outer_k, outer_v) in people_dictionary.items()
}

print(people_names)
#=> {'member_01': {'Sal'}, 'member_02': {'Tom'}}
```

Double each value in the dictionary

Double each value in the dictionary only if the value is an even number

Identify odd and even entries

Similarly, dictionaries can be nested and thus their comprehensions can be nested as well

Classes, Modules, & Packages

Everything in Python is an object and all objects have a class type; each of those class types in turn have methods that can be called on them; **num_var** is based on the list class class type therefore it inherits all methods for the list class

```
my_value = True
type(my_value)
#=> <class 'bool'>

my_var = 33
type(my_var)
#=> <class 'int'>

numb_var = [1, 6, 3]
type(num_var)
#=> <class 'list'>

people_ages = {'ron':7, 'bob':12, 'tom':12}
type(people_ages)
#=> <class 'dict'>

another = "Qwerty"
type(another)
#=> <class 'str'>
```

the init function

The init function can be hardwired like **this** but is usually built to accept **instance variables**

```
class Planet:

  def __init__(self):
    self.name = 'Hardwired Planet Name'
    self.radius = 3.14
    self.gravity = 5.5
    self.system = "Hoth System"
```

files shown in the sections below are available here: **github.com/Richard-Burd/python-3-sandbox**

class name is capitalized & followed by ":"

This is where class attributes are defined

These are the instance attributes

.python-3-sandbox/lessons/space/**planet.py**

```
class Planet:

  shape = 'round'

  def __init__(self, name, radius, gravity, system):
    self.name = name
    self.radius = radius
    self.gravity = gravity
    self.system = system

  def orbit(self):
    return f'{self.name} is orbiting in the {self.system}'

  @staticmethod
  def spin(speed = '2000 miles per hour'):
    return f'The planet spins and spins at {speed}'

  @classmethod
  def commons(cls):
    return f'All planets are {cls.shape} because of gravity'
```

This function defines the class attributes & it takes in a **self** property

static method

cls

like 'self' but for the class

class method

These are both instance methods, so they both must take in the **self** parameter

The class attributes can be called by the class itself or a class instance

.python-3-sandbox/lessons/space/**__init__.py**

You don't need anything in this file; it's existence alone is enough to tell Python that the contents in this directory called **space** are a package of modules

.python-3-sandbox/lessons/space/**__pycache__**

This directory gets generated automatically when you create the folder above to setup a module

Static methods have no access to class (cls) or class intances (self)

space is the folder name, **planet** is the name of the class file; this import is enabled by the **__init__.py** file that is in the **space** directory

.python-3-sandbox/lessons/**classes.py**

```
from space.planet import Planet
from space.calc import planet_mass, planet_vol #use "import *" to import all

naboo = Planet('Naboo', 300000, 8, 'Naboo System')
naboo_mass = planet_mass(naboo.gravity, naboo.radius)
naboo_vol = planet_vol(naboo.radius)

print(f'{naboo.name} has a mass of {naboo_mass} and a volume of {naboo_vol}')
#=> Naboo has a mass of 1.0794602698650675e+22 and a volume of 377040000000.0

print(f'default: {naboo.spin()}')
#=> default: The planet spins and spins at 2000 miles per hour

print(f'custom: {naboo.spin("100 mph")}') same as: print(f'custom: {Planet.spin("100 mph")}')
#=> custom: The planet spins and spins at 100 mph

print(naboo.commons())
#=> All planets are round because of gravity
```

a class instance called 'naboo' is created above in classes.py

.python-3-sandbox/lessons/space/**calc.py**

```
def planet_mass(gravity, radius):
    mass = ( gravity*radius**2 ) / ( 6.67*10**-11 )
    return mass

def planet_vol(radius):
    vol = ( 4*3.142*radius**2 ) / 3
    return vol
```

this is a module

```
print(naboo.shape)
print(Planet.shape)
#=> 'round'
#=> 'round'
```

Console Inputs & Strings

User inputs are always accepted as strings (e.g. "2") unless they are type-casted with **int()** to become integers

print()

int()

./project-path/**myScript.py**

type casting

```
radius = input("Enter the radius of your circle (m):")

area = 3.142 * int(radius) ** 2
print('The area of your circle is:', int(area))
```

This will get printed out and prompt the user for an input

```
user@ubuntu ~/project-path $ python3 myScript.py
=> Enter the radius of your circle (m): 4
=> The area of your circle is: 50
```

Up here we broke out of the string
But now let's look at some string formatting

.format()

```
num1 = 3.1425
num2 = 10.2903

print('The first variable is {0} and the second one is {1}'.format(num1, num2))
#=> The first variable is 3.1425 and the second one is 10.2903

print('The first variable is {0:3} and the second one is {1:.4}'.format(num1, num2))
#=> The first variable is 3.14 and the second one is 10.29

print(f'The first variable is {num1:.2f} and the second one is {num2:.4f}')
#=> The first variable is 3.14 and the second one is 10.2903
```

0 in the list
1 in the list
list = [num1, num2]

f-strings

We could specify **3** or even **4** digits in the number (precision)

These are called F-Strings & they don't require ".**format**()"

Or we could add the little **f** for float to specify the deceimal number we want to see

Ranges

In Python, ranges generate a list of numbers for us that we can then use to iterate over in **for** loops.

```
for n in range(5):
  print(n)
#=> 0 #=> 1 #=> 2 #=> 3 #=> 4

for n in range(5,10):
  print(n)
#=> 5 #=> 6 #=> 7 #=> 8 #=> 9

for n in range(20,300, 80):
  print(n)
#=> 20 #=> 100 #=> 180 #=> 260
```

this will go up to but not including 5

this will start from and include 5 and go through but not include 10

this will **start** from and include 20 and go **through** but not **include** 300 in intervals of 80

len()

```
names = ['bo', 'bob', 'tom', 'dan']

for n in range(len(names)):
  print(n + 1, ".)", names[n].capitalize())
#=> 1.) Bo #=> 2.) Bob #=> 3.) Tom #=> 4.) Dan

for n in range(len(names) -1, -1, -2):
  print(n + 1, ".)", names[n])
#=> 4.) dan
#=> 2.) bob
```

this **len()** method finds the length of the **names** list and cycles throught the range for each element in that **names** list

this -1 is the last item in the list and the **start** position, this -1 is the position right before the start of the list because this value tells us the looping **end** point and it is an up-to-but-not-included value, finally, this -2 is the **increment** amount, and it is negative

Dictionary Iteration

code shown in the section below is available here:
github.com/Richard-Burd/python-3-sandbox/dictionary_iteration.py

```
users = {
  'user-1': {
    'name':"Sam",

    'work-history' : {
      'job-1' : 'server',
      'job-2' : 'dentist'
    },

    'bio': "I was a server
            and a dentist"
  },

  'user-2': {
    'name':"Dan",

    'work-history' : {
      'job-1' : 'plumber',
      'job-2' : 'doctor'
    },

    'bio': "I was a plumber
            and a doctor"
  },
}
```

```
def work_history_profession_finder(users, profession):
  for user_key, user_value in users.items():
    for category_key, category_value in user_value.items():
      if category_key == 'bio' and profession in category_value:
        print(f'{user_value.get("name")} was a {profession}')
work_history_profession_finder(users, 'doctor')
#=> Dan
```

for loops can be used to find values within nested dictionaries

and in

```
def bio_profession_finder(users, profession):
  for user_key, user_value in users.items():
    for category_key, category_value in user_value.items():
      if category_key == 'work-history':
        for job_key, job_value in category_value.items():
          if profession in job_value:
            print(f'{user_value.get("name")} was a {profession}')
bio_profession_finder(users, 'dentist')
#=> Sam
```

```
if True | False:
  print("truthy")
else:
  print("falsey")
#=> truthy

if False | True:
  print("truthy")
else:
  print("falsey")
#=> truthy

if False | False:
  print("truthy")
else:
  print("falsey")
#=> falsey
```

JavaScript & Ruby pipes (||) are replaced with a single pipe in Python

Maps

Maps are a way to take a list, apply some kind of function to each item within that list, and return a new list with the changes made by the function to each item in the list

random

```
from random import shuffle

words = ['ron', 'cat', 'pat']

anagrams = []

def jumble(word):
  anagram = list(word)
  shuffle(anagram)
  return ''.join(anagram)
```

we need the random module from the Python Standard Library for this example:
https://docs.python.org/3/library/

list

the **list()** method takes a string and makes each character its own string in a new list like: ['r', 'o', 'n']

join

```
''.join(['r', 'o', 'n'])
#=> 'ron'

'-'.join(['r', 'o', 'n'])
#=> 'r-o-n'
```

the **join()** method will take elements of a list and put them in the same string

map (function used, list being operated on)
'jumble' 'words'

Standard Method

```
for word in words:
  anagrams.append(jumble(word))

print(anagrams)
#=> ['orn', 'cta', 'atp']
```

map()

Vanilla Map Method

```
print(map(jumble, words))
#=> <map object at 0x7faa43897128>
```

This doesn't quite work because it's mapping the result onto an **unusable object**; to make it readable, it myst be typecasted into a **list**

Comprehension Method

```
print([jumble(word) for word in words])
#=> ['nor', 'tca', 'tap']
```

Working Map Method

```
print(list(map(jumble, words)))
#=> ['nor', 'tca', 'tap']
```

Filters

The **filter** method is used to determine if a specified condition is true or false for each element in a given list; if true, the element remains in the filtered list, if not, it is dropped; The example here filters out bad grades.

```
grades = ['A', 'B', 'F', 'C', 'F', 'D', 'A']
```

filter()

Using the Filter Method

```
def remove_bad_grades(my_grade):
  return my_grade != 'F' and my_grade != 'D'

print(filter(remove_bad_grades, grades))
#=> <filter object at 0x7f0aaf84b780>

print(list(filter(remove_bad_grades, grades)))
#=> ['A', 'B', 'C', 'A']
```

Using a For Loop

```
filtered_grades = []
for grade in grades:
  if grade != 'F' and grade != 'D':
    filtered_grades.append(grade)
print(filtered_grades)
#=> ['A', 'B', 'C', 'A']
```

filter (testing function, list being operated on)
'remove_bad_grades' 'grades'

Using Comprehension

```
print([grade for grade in grades if grade != 'F' and grade != 'D'])
#=> ['A', 'B', 'C', 'A']
```

comprehension is still the shortest way to go

Collections - Deque

all code shown in this section is available here:
github.com/Richard-Burd/python-3-sandbox/timfiles/collections_deque.py

Why use a deque over a traditional list? because it's faster in terms of adding items to the beginning and end of the list, but if you're going to want to randomally access elements within the container, then it's better to use a traditional list.

```
from collections import deque

d = deque("hello")
print(d)
#=> deque(['h', 'e', 'l', 'l', 'o'])

d.append('4')
d.append('5')
d.appendleft('2')
d.appendleft(1)
d.appendleft(True)
print(d)
#=> deque([True, 1, '2', 'h', 'e', 'l', 'l', 'o', '4', '5'])

d.pop()
d.popleft()
print(d)
#=> deque([1, '2', 'h', 'e', 'l', 'l', 'o', '4'])

d.clear()
print(d)
#=> deque([])

d.extend('456')
d.extend('hello')
d.extendleft('123')
print(d)
#=> deque(['3', '2', '1', '4', '5', '6', 'h', 'e', 'l', 'l', 'o'])
```

the **Deque** must be imported

appendleft

it takes each element of a string and makes it an element of a list-like data structure

pop

items can be added to the **front** or **back** of a deque

clear

this will destructively clear all contents of the deque

extend

the **extend()** method takes in anything iterable, such as a list or string, ant puts it at the end of the **deque**

```
x = deque("123456")
print(x)
#=> deque(['1', '2', '3', '4', '5', '6'])

x.rotate(-1)
print(x)
#=> deque(['2', '3', '4', '5', '6', '1'])

x.rotate(2)
print(x)
#=> deque(['6', '1', '2', '3', '4', '5'])

w = deque("Interlochen", maxlen=5)
print(w)
#=> deque(['o', 'c', 'h', 'e', 'n'], maxlen=5)

print(w.maxlen)
#=> 5

w.extend([1, 2, 3])
print(w)
#=> deque(['e', 'n', 1, 2, 3], maxlen=5)
```

The **rotate()** method shifts the order of items as shown on the left

rotate

The **maxlen()** method limits the number of items in the deque
NOTE: you cannot reassign this **maxlen** value after it's initially declared

maxlen

you can only add to the deque with the **extend()** method, but that will still mantain the original maxlen of 5

{Sets} [Lists] & Sorting

all code shown in this section is available here: **github.com/Richard-Burd/python-3-sandbox/sets.py**

Python lists are similiar to JavaScript & Ruby arrays whereas Python sets are essentially Python dictionaries with only keys, and no values. Every element in a set must be immutable but the set itself is mutable

```
my_new_var = {}
print(type(my_new_var))
#=> <class 'dict'>

number_list = [1, 6, 3, 2, 4, 4, 1, 0, 7, 2]
print(sorted(number_list))
#=> [0, 1, 1, 2, 2, 3, 4, 4, 6, 7]

number_set = {1, 6, 3, 2, 4, 4, 1, 0, 7, 2}
print(sorted(number_set))
#=> [0, 1, 2, 3, 4, 6, 7]

people_list = ['ron', 'Bob', 'Tom', 'Dan', 'cat', 'alf', 'ron']
print(sorted(people_list))
#=> ['Bob', 'Dan', 'Tom', 'alf', 'cat', 'ron', 'ron']

people_set = {'ron', 'Bob', 'Tom', 'Dan', 'cat', 'alf', 'ron'}
print(sorted(people_set))
#=> ['Bob', 'Dan', 'Tom', 'alf', 'cat', 'ron']

print(set(number_set))
#=> {0, 1, 2, 3, 4, 6, 7}

print(set(people_list))
#=> {'Tom', 'alf', 'ron', 'cat', 'Dan', 'Bob'}
```

this will be a dictionary, not a set because that is what an empty {} defaults to in Python

sorted() method sorts numbers & strings…

…it will remove duplicates in a set

…it will order items with capital letters first

set() will return a set and thus, remove duplicates in a list of strings, but it will not order the elements so the return value can change each time

https://stackoverflow.com/questions/2831212/python-sets-vs-lists/2831242
Lists are slightly faster than sets when you just want to iterate over the values. Sets, however, are significantly faster than lists if you want to check if an item is contained within it. They can only contain unique items though. It turns out tuples perform in almost exactly the same way as lists, except for their immutability.

```
my_set = {5, 'a', '3', 6, 'Q', 2}
print({elem for elem in my_set})
#=> {2, 5, 6, '3', 'Q', 'a'}

a_new_set = set("abcdef")
print(a_new_set)
#=> {'f', 'd', 'b', 'c', 'e', 'a'}

a_new_set.add("my string")
print(a_new_set)
#=> {'e', 'a', 'f', 'd', 'c', 'my string', 'b'}

mutable_thing = [1, 2, 7]
a_new_set.add(mutable_thing)
#=> TypeError: unhashable type: 'list'

del a_new_set
```

It is possible to do comprehension on sets

Sets are mutable…

…however, elements of a set cannot be mutable objects

NOTE: this order will change each time the code is ran because there is no true order to a set, unlike a list

```
people_ages = {'ron':7, 'bob':12, 'tom':12, 'dan':1, 'cat':1, 'alf':1}
print(set(people_ages.values()))
#=> {1, 12, 7}
print(set(people_ages.keys()))
#=> {'alf', 'tom', 'cat', 'dan', 'ron', 'bob'}

def age_count(dictionary):
  ages = list(dictionary.values())
  for age in set(ages):
    number = ages.count(age)
    print(f'There are {number} people aged {age}')

age_count(people_ages)
#=> There are 3 people aged 1
#=> There are 2 people aged 12
#=> There are 1 people aged 7
```

keys values

here we are typecasting the ages into a set using the **set()** method so that we do not have duplicate ages down here

Try & Accept

all code shown in this section is available here:
github.com/Richard-Burd/python-3-sandbox/timfiles/try_&_accept.py

Try & accept lets you run code that actually would crash if it is falsey; Here I want my string to only be numbers:

```
print("Enter a username that consists only of numbers")
text = input("Username: ")

try:
  number = int(text)
  print(f'So, {number} is in fact a legit username!')
except:
  print("Invalid username, access denied :(")
```

Here the text is typecasted into an integer

The **except** block will run if the **try** block of code is either falsey, or crashes

Functions & Variable Scope

Python uses colons to start a function body

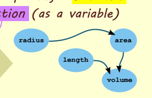
```
def greet(name = "Moe", time = "12:30pm"):
  print(f"Hey {name} it's {time} right now!")

greet(name = "Ron")
#=> "Hey Ron it's 12:30pm right now!"
```

The function body must be indented or Python will not compile

To overide default values, **specify the variable** that will have the default overidden

```
def area(radius):
  return 3.142 * radius * radius

def volume(area, length):
  print(area * length)

radius = int(input('enter a radius:'))
length = int(input('enter a length'))

volume(area(radius), length)
```

Here we are passing a **function** into a **function** (as a variable)



https://repl.it/@Richard_Burd/Variable-Scope#main.py

```
my_name = "Dog"

def parent_scope():
  my_name = "Mom"
  print(f"I see {my_name} in the parent scope.")
  def child_scope():
    my_name = "Kid"
    print(f"I see {my_name} in the child scope.")
    def pet_scope():
      global my_name
      print(f"I see {my_name} in the pet scope.")

    # call the child function:
    pet_scope()

  # call the child function:
  child_scope()

# call the parent function:
parent_scope()

#=> I see Mom in the parent scope.
#=> I see Kid in the child scope.
#=> I see Dog in the pet scope.
```

variables can be redefined in a lower scope but still retain their original value in the higher scope; the global value can be called with the **global** keyword in a lower scope

global

Advanced Overview Features

Most of the code shown in this section is available here:
github.com/Richard-Burd/python-3-sandbox/timfiles/expert.py

Python is compiled into bytecode before it is interpreted. Compilers take high-level code and translate it into a lower-level. An interpreter takes some kind of code, in our case bytecode, and interprets & runs that code. This is unique to Python because it is a compiled language, here we have a class with an undefined 'bark' method:

```
class Dog:
  def __init__(self):
    self.bark()
```

This is unique to Python because it is a compiled language, here we have a class with an undefined 'bark' method that has not yet been defined; If I run the code at this point there will be no errors. In other languages, the compiler would detect this error and tell you to define what 'bark' is, but here, this bit of code is executed at runtime instead of compile time. All the compiler does for us is translate the Python into bytecode, and it does not always check to see if the code is actually valid. Thus, the error above is said to be 'only caught at runtime' and not at compile time.

Let's look at another example in which the compiler doesn't care if the code is valid or not, so long as you have valid syntax

```
def make_class(x):
  class Cat:
    def __init__(self, name):
      self.name = name

    def print_value(self):
      print(x)

  return Cat

cls = make_class(10)
print(cls)
#=> <class '__main__.make_class.<locals>.Cat'>

d = cls("Timmy")
d.print_value()
#=> 10

print(d.name)
#=> Timmy
```

We can define a class inside a function because that is how Python works; we can nest classes as deep as we want.

This returns the class **Cat** and not an instance of the class; the class is being created and stored in memory

This **cls** variable is actually a class, so it's another name for **Cat**

Here we're calling the class method

the **name** of this instance of the Cat class is "Timmy"

```
for i in range(10):
  def show():
    print(i*2)

  show()
show()
```

Functions can be put inside of a **for** loop

This will run each time the loop runs

This will only run once on the final item in the range, but it is aware of the existence of **show()** inside a deeper scope it is not a part of, but **show()** must be declared on a line ABOVE wherever it is being called or it will not run and will generate an **'is not defined'** error

all code shown in this tiny subsection is available here:
github.com/Richard-Burd/python-3-sandbox/timfiles/nested_function_calls.py

```
def outer_function(x):
  def inner_function(x):
    print(f'get inside with a {x}!')

  return inner_function

zelda = outer_function("key")
zelda("key")
#=> get inside with a key!

print(id(zelda))
#=> a thing like "140175176829808"

import inspect

print(inspect.getmembers(zelda()))
#=> [('__bool__', <method-wrapper '__bool__' of NoneType object at 0x9d4380>),et.al]

print(inspect.getsource(zelda))
#=>     def inner_function(x):
#=>         print(f'get inside with a {x}!')
```

A function inside of a function must be accessed by nesting the function call like this

inspect

Everything in Python is an object so each thing has its own unique memory address

getsource

The **inspect** module can show us some pretty cool things because of the fact that all of our Python objects are live; **here** we use the **getsource()** method to get the sourcecode of a specific method, function, class, or other object

Zip Function

Reference: **https://www.w3schools.com/python/ref_func_zip.asp**

all code shown in this section is available here:
github.com/Richard-Burd/python-3-sandbox/zip.py

The **zip()** function returns a zip object, which is an iterator of tuples where the first item in each passed iterator is paired together, and then the second item in each passed iterator are paired together etc. If the passed iterators have different lengths, the iterator with the least items decides the length of the new iterator.

.python-3-sandbox/**zip.py**

```
names = ("John", "Charles", "Mike")
ages = (25, 22, 34, 57)

output = zip(names, ages)

print(tuple(output))
#=> (('John', 25), ('Charles', 22), ('Mike', 34))

print(dict(output))
#=> {'John': 25, 'Charles': 22, 'Mike': 34}
```

use the **tuple()** function to display a readable version of the result…

…or use the **dict()** function instead, but you can only run one or the other in the same script on the same object

Collections

Standard Python Containers
1.) list
2.) set
3.) dictionary (dict)
4.) tuple ← this one is immutable

Collections Module Containers
1.) counter
2.) deque
3.) namedTuple
4.) OrderedDict
5.) defaultDict

.python-3-sandbox/**test.py**

```
from collections import Counter, deque, namedtuple, orderedDict

a = Counter('gallad')
print(a)
#=> Counter({'a': 2, 'l': 2, 'g': 1, 'd': 1})

d = deque("hello")
print(d)
#=> deque(['h', 'e', 'l', 'l', 'o'])

Point = namedtuple('Point', 'x y z time')
newP = Point(3, 4, 5, '12:00pm')
print(newP)
#=> Point(x=3, y=4, z=5, time='12:00pm')
```

The Python data types above must be imported via their commensurate module in order to be used as shown on the left

Collections - Counter

all code shown in this section is available here:
https://github.com/Richard-Burd/python-3-sandbox/timfiles/collections_counter.py

This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers

```
from collections import Counter

a = Counter('gallad')
print(a)
#=> Counter({'a': 2, 'l': 2, 'g': 1, 'd': 1})

a = Counter(['a', 'a', 'b', 'c', 'c'])
print(a)
#=> Counter({'a': 2, 'c': 2, 'b': 1})

a = Counter({'a':1, 'b':2})
print(a)
#=> Counter({'b': 2, 'a': 1})

a = Counter(cats=4, dogs=7)
print(a['cats'])
#=> 4

print(a['lizards'])
#=> 0

print(list(a.elements()))
#=> ['cats', 'cats', 'cats', 'cats', 'dogs', 'dogs', 'dogs', 'dogs', 'dogs', 'dogs', 'dogs']

b = Counter({'x':2, 'y':7, 34:8, 3:1, 'z':45})
print(list(b.elements()))
#=> ['x', 'x', 'y', 'y', 'y', 'y', 'y', 'y', 'y', 34, 34, 34, 34, 34, 34, 34, 34, 3, 'z', et.al]

print(f'The two most common key-value pairs are {b.most_common(2)}')
#=> The two most common key-value pairs are [('z', 45), (34, 8)]

b.subtract('z')
print(b)
#=> Counter({'z': 44, 34: 8, 'y': 7, 'x': 2, 3: 1})
```

the **Counter** must be imported

a string as a variable will return a letter count

a list will return a dictionary showing how many times each item occurs in the dictionary

a dictionary will return a sorted dictionary

keys can be non-string variables…

…but must be called with a string

a key that doesn't exist will return 0

listing elements will return a list of items

Counter keys can be different data types

most_common

Counter has a **most_common** method for returning items by **number of most common** items

you can **subtract** keys that are strings but not ones that are integers like **these two**

```
d = ['x', 'x', 'x', 'x']
b.update(d)
print(b)
#=> Counter({'z': 44, 34: 8, 'y': 7, 'x': 6, 3: 1})

b.clear()
print(b)
#=> Counter()

a = Counter({'x':1, 'y':2, 'z':3})
b = Counter({'x':10, 'y':10, 'z':10})

print(a + b)
#=> Counter({'z': 13, 'y': 12, 'x': 11})

print(b - a)
#=> Counter({'x': 9, 'y': 8, 'z': 7})

print(a - b)
#=> Counter()

print(b & a)
#=> Counter({'z': 3, 'y': 2, 'x': 1})

print(a | b)
#=> Counter({'x': 10, 'y': 10, 'z': 10})
```

```
d = {'x':4}
```

Counter keys can be updated like **this** or like **this**

the **Counter** can be cleared of all its contents

When you subtract elements on a counter, it will not show values of 0 or negative values

Here we say that **b** is **"intersecting"** with **a** and this gives the lowest common values for the items in the counter, in this case, **z** has a value of 3 (in **Counter a** above) and a value of 10 (in **Counter b** above) so since 3 is the smallest, that is the intersect value

intersection & union of Counters

The opposite of intersecting is called **"union"** which is shown here: These are the maximum values between the two Counters **a** and **b** above

(x & y)

(x | y)

Reading Files

Python allows you to open up files and read them then do something with those files

.python-3-sandbox/lessons/**read.py**

```
ipsum_file = open('files/ipsum.txt')

for my_line in ipsum_file:
  print(my_line)

for line in ipsum_file:
  print(line.rstrip())

ipsum_file.seek(6)
#=> Lorem ipsum dolor sit...(et.al.)
#=> [ipsum dolor sit...(et.al.)]

selected_text = ipsum_file.read(5)
print(selected_text)
#=> ipsum

ipsum_file.close()
```

this opens an external file at:
.python-3-sandbox/lessons/files/**ipsum.txt**

this iterates through each line of the file & reads each one

the **rstrip()** method removes empty spaces between lines

this starts at the 6th character in the file and returns it in list brackets **"[]"**

this starts from the **seek()** method starting point above **(6)** and reads **5** characters of the file

the file should always be closed to prevent any performance penalties

```
def sequence_filter(line):
  return '>' not in line

with open('files/dna_sequence.txt') as dna_file:
  lines = dna_file.readlines()
  print(list(filter(sequence_filter, lines)))
```

this checks to see if the **">"** character is in the line or not

using **with open() as:** is generally better than using the seperate **open() & close()** statements shown above; the file remains open while code beneath is **indented** and closes when the code indentation ends

.python-3-sandbox/lessons/files/**dna_sequence.txt**

```
>SEQUENCE_1
MTEITAAMVKELRESTGAGMMDCKNALSETNG...et.al
>SEQUENCE_2
ARGLVVKVVSDDFTIAAMRPSYLSYEDLDMTF...et.al
>SEQUENCE_3
IPQFASRKQLSDAILKEAEEKIKEELKAQGKP...et.al
>SEQUENCE_4
MGQFYVMDDKKTVEQVIAEKEKEFGGKIKIVE...et.al
```

this finds all instances of the **">"** character in the **dna_sequence.txt** file and filters them out of the printed list in the console

Lambdas

Lambda expressions (or lambda functions) are similar to anonymous functions in JavaScript; they are suitable for situations in which you're only gonna call the function once.

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

def square(n):
  return n*n

print(list(map(square, nums)))
#=> [1, 4, 9, 16, 25, 36]
```

This is a standard function that will square a number with a map function typecasted into a list

lambda

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

print(list(map(lambda n: n*n, nums)))
#=> [1, 4, 9, 16, 25, 36]
```

The lambda will automatically return the result to the right of the ":" without a **return** statement

```
lambda x, y, z: n*n
```

lambda (first variable, second variable: calculation)
'n' (not in example) 'n*n'

You could pass in multiple arguments into a lambda like this

Collections - NamedTuple

all code shown in this section is available here:
github.com/Richard-Burd/python-3-sandbox/timfiles/collections_namedtuple.py

The main difference between a regular tuple and a named tuple is that with a named tuple you can access things by element and it's a lot nicer to read in your program

```
from collections import namedtuple

Point = namedtuple('Point', 'x y z time')

newP = Point(3, 4, 5, '12:00pm')
print(newP)

#=> Point(x=3, y=4, z=5, time='12:00pm')
```

the **namedtuple** must be imported

subclass constructor

instantiator

a **namedtuple** subclass name must be a **capitalized string** and the item names are declared in the **second string** with each item seperated by a space

```
Coor = namedtuple('Coordinates', ['lat', 'long'])
new_coor = Coor(20.213, 499.340)
print(new_coor)
#=> Coordinates(lat=20.213, long=499.34)

Data = namedtuple('Data_Set', {'x':0, 'y':0, 'z':0})
new_data = Data(10, 20, 30)
print(new_data)
#=> Data_Set(x=10, y=20, z=30)

print(new_data.x)
#=> 10

print(new_data[0])
#=> 10
```

the items can be put in a list like this

the items can also be stored in a dictionary

Named tuples allow for the use of dot notation, but regular tuples do not

_asdict()

with Named tupels, items can be found by inded as well

_replace()

values can be converted to a dictionary with the **_asdict()** method

```
print(new_data._asdict())
#=> OrderedDict([('x', 10), ('y', 20), ('z', 30)])

print(new_data._fields)
#=> ('x', 'y', 'z')

my_new_value = new_data._replace(y=88)
print(my_new_value)
#=> Data_Set(x=10, y=88, z=30)

p2 = Data._make(['a', 'b', 'c'])
print(p2)
#=> Data_Set(x='a', y='b', z='c')
```

_fields()

We can print out the keys with **the _fields()** method

we can replace the value of a specified key, with the **_replace()** method, but this is not destructive, in other words, we need to assign a new value to the operation because we cannot change the original namedtuple in this way.

_make()

The **_make()** method can be used to create a new instance of the Data class and this new instance will have the specified values passed into the **_make()** method. NOTE: the values are now strings, not integers as in the original declaration above, because we can change the data type

Queue Basics

all code shown in this section is available here:
github.com/Richard-Burd/python-3-sandbox/misc_tutorials/PyMoondra_Threading_Series/queue_basics.py

There are 3 types of queues: FIFO, LIFO, and Priority; FIFO is the default type when no other type is specified.

First In First Out (FIFO)

```
import queue

q = queue.Queue()

q.put(5)
q.put(1)
q.put("three")
q.put("7")

print(q.get())
#=> 5

print(q.get())
#=> 1

print(q.empty())
#=> False

while not q.empty():
  print(q.get(), end = ' ^ ')
#=> three ^ 7 ^
```

The **.get()** method returns the next item in the queue to be retrieved; in this case, it is 5 because the 5 was the **first in,** and is therefore the first out - the **1** is next in line and thus, the **.get() method is destructive**

The **.empty()** method returns a boolean value

get empty

A while loop can return all existing values, we use **empty()** instead of **get()** because the latter will cause our computer to lock up when there are no more items to actually "**get()**" in the queue - to solve that problem we can use an advanced method called **threading** or the **empty()** method as shown

Last In First Out (LIFO)

```
q2 = queue.LifoQueue()

q2.put("Ron")
q2.put("Tom")
q2.put("Dan")

while not q2.empty():
  print(q2.get(), end = ' ^ ')
#=> Dan ^ Tom ^ Ron ^
```

LifoQueue

This is the last item put in the queue, and so it is the **first** item to go out when the **get()** method is called on the queue

Priority Queue (Integers)

```
import time

q3 = queue.PriorityQueue()

q3.put(1)
q3.put(3)
q3.put(4)
q3.put(2)

for i in range(q3.qsize()):
  print(q3.get())
#=> 1
#=> 2
#=> 3
#=> 4
```

PriorityQueue

If the queue items are simply integers, then the **get()** method will return the integers from **smallest** to **largest**

Priority Queue (Keys+Values)

```
q4 = queue.PriorityQueue()

q4.put((1, 'Priority 1'))
q4.put((3, 'Priority 3'))
q4.put((4, 'Priority 4'))
q4.put((2, 'Priority 2'))

for i in range(q4.qsize()):
  print(q4.get())
#=> (1, 'Priority 1')
#=> (2, 'Priority 2')
#=> (3, 'Priority 3')
#=> (4, 'Priority 4')
```

If you need to assign a priority to a data type, let's say, to a string, then you use a tuple for each use of the **put()** method wherein the **first tuple item is an integer** (representing the priority value) and the **second tuple item** is the string (or some other data type) - the first tuple item must always be an integer

Writing Files

When we open up a file, it is read-only by default unless the **open()** method takes in ta second variable, **'w'** that permits writing to the file or an **'a'** that permits appending the file

.python-3-sandbox/lessons/**write.py**

```
with open('files/write.txt', 'w') as write_file:
  write_file.write("I will be written only once in the file and wipe out all previous content.")

with open('files/write.txt', 'a') as write_file:
  write_file.write("\nI am being amended to the string above.")

lahja_arabiya_quotes = [
"\n1.) Marhaba ya a7",
"\n2.) 3ni alek",
"\n3.) Saba7 al 7er"
]

with open('files/write.txt', 'a') as write_file:
  write_file.writelines(lahja_arabiya_quotes)
```

the **'w'** method will open up the file, look at what is inside, and override whatever is inside it; that means this string will appear only once in that file and everything else in that file will be deleted. The **'a'** method on the other hand will only add text to the end of the file and leave the existing text content alone

.python-3-sandbox/lessons/files/**write.txt**

write() writelines()

```
I will be written only once in the file and wipe out all previous content.
I am being amended to the string above.
1.) Marhaba ya a7
2.) 3ni alek
3.) Saba7 al 7er
```

The **writelines()** method expects some kind of (Python) list that it will go through as print each element within it

Dunder Magic

all code shown in this section is available here:
https://github.com/Richard-Burd/python-3-sandbox/timfiles/dunder_magic.py

```
class Dog:
  def __init__(self, name):
    self.name = name

d = Dog("Fido")

print(d)
#=> <__main__.Person object at 0x7f60d1aa5780>
```

By default we get the memory address location of this object because we have not told the object what to do when we print it

__repr__

```
class Person:
  def __init__(self, name):
    self.name = name

  def __repr__(self):
    return f"My Person({self.name})"

  def __mul__(self, x):
    if type(x) is not int:
      raise Exception("invalid argument, must be an integer!")

    self.name = "^" + ("*" + self.name + "*") * x + "^"

  def __call__(self, y):
    print("called this function", y)

  def __len__(self):
    return len(self.name)
```

To get meaningful information, we need to implement a 'Dunder' method AKA a 'magic' method

__mul__

__call__

The **__mul__** data model method will tell Python how to handle multiplication operations on an object that is of the data type **Person,** e.g. an instance of the **Person** class

__len__

raise Exception

Whenever this Person class is called as if it were a function, this **__call__** method will handle that call for the Person class

These are all data model methods

data model methods

```
p = Person("Tim")
p * 4
print(p)
#=> My Person(^*Tim**Tim**Tim**Tim*^)

p(22)
#=> called this function 22

psn = Person("four")
print(len(psn))
#=> 4
```

The **__len__** data model method will tell Python what to do when the length (**len**) is called on the **Person** class; in this example, the string "four" has **4** items

Queue

```
from queue import Queue
import inspect

q1 = Queue()
print(q1)
#=> <queue.Queue object at 0x7fbb5ddaf240>

print(inspect.getsource(Queue))
#=> <queue.Queue object at 0x7fd87ebac240>
#=> class Queue:
#=>     '''Create a queue object with a given maximum
#=>
#=>     If maxsize is <= 0, the queue size is infinite.
#=>     '''
#=> etc...this goes on forever
```

this just returns an object address

OK let's now let's set things up so that when we print the queue by calling the **print** method on the declaration, we get something meaningful to print in the console instead of this

```
from queue import Queue as q
import inspect

class Queue(q):
  def __repr__(self):
    return f"Queue size is: {self._qsize()}"

  def __add__(self, item):
    self.put(item)

  def __sub__(self, item):
    self.get()

q2 = Queue()

q2 + "a"
q2 + 8
q2 + True
q2 - [2, 3, 1]
print(q2)
#=> Queue size is: 3

print(list(q2.queue))
#=> [8, True]
```

__add__

__sub__

The **__add__** method will fire anytime something is added to the queue

The **__sub__** method will fire anytime something is subtracted from the queue

Without the **__add__** method above, this would generate this error:
TypeError: unsupported operand type(s) for +: 'Queue' and 'int'

In **this case,** it doesn't matter what value you put here, because this queue is (by default) a **First-In-First-Out (FIFO)** queue, and the first item ("a") will be gotten {**get()**} and then subtracted from the queue because the **__sub__** method is being called to perform a subtraction operation

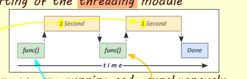
Threading

all code shown in this section is available here:
github.com/Richard-Burd/python-3-sandbox/misc_tutorials/corey_threading/vid_1.py

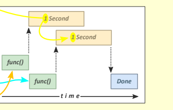
Diagrams are from this video here on Python threading by Corey Schafer:
https://www.youtube.com/watch?v=IEEhzQoKtQU

Threading enables concurrent code execution & requires the importing of the **threading** module

```
import time

start = time.perf_counter()

def do_something():
  print('Sleeping 1 second...')
  time.sleep(1)
  print('Done Sleeping...')

do_something()
do_something()

finish = time.perf_counter()

print(f'Finished synchronous processes in {round(finish-start, 2)} second(s)')

import threading

t1 = threading.Thread(target=do_something)
t2 = threading.Thread(target=do_something)

start2 = time.perf_counter()

t1.start()
t2.start()

t1.join()
t2.join()

finish2 = time.perf_counter()

print(f'Finished concurrent processes using threading in {round(finish2-start2, 2)} second(s)')
```



Here we are running code synchronously
first we run **this code** from start to finish
then we run **this code** from start to finish



Here we are running code concurrently; the **2nd function** fires after the **1st function** fires, but the code is not ran simultaneously

The **join()** method will ensure that **t1** and **t2** finish running before moving onto the next lines where the **finish time is calculated** and **this statement** gets printed out

threading.Thread (target=the function we want to run)
'do_something'

join

Downloading Files

.python-3-sandbox/projects/**download_img.py**

```
import urllib.request

def download_image(url, file_path, file_name):
  full_path = file_path + file_name + '.jpg'
  urllib.request.urlretrieve(url, full_path)

url = input('Enter img URL to download:')
file_name = input('Enter file name to save as:')

download_image(url, 'images/', file_name)
```

We need this module to be imported in order to download

These receive input from the (CLI) user & store their answers as variables

.python-3-sandbox/projects/**images**

Create this folder and leave it empty; images will be download to it

Running thes commands in the bash prompt will generate an **Airplane.jpg** file inside the **images** folder

```
user@ubuntu ~/python-3-sandbox/projects $ python3 download_image.py
Enter img URL to download: https://i.imgur.com/K4h9aD4.jpg
Enter file name to save as: Airplane
```

References:

Python 3 Tutorial for Beginners by The Net Ninja
https://www.youtube.com/playlist?list=PL4cUxeGkcC9idu6GZ8EU_5B6WpKTdYZbK

Python Programming Tutorials by Tech With Tim
https://www.youtube.com/playlist?list=PLzMcBGfZo4-mFu00qxl0a67RhjjZj3jXm

Intermediate Python Tutorials by Tech With Tim
https://www.youtube.com/playlist?list=PLzMcBGfZo4-nhWva-6OVh1yKWHBs4o_tv

Expert Python Tutorials by Tech With Tim
https://www.youtube.com/playlist?list=PLzMcBGfZo4-kwmIcMDdXSuy_wSqtU-xDP

Mastering Python by Tech With Tim
https://www.youtube.com/watch?v=p15xzjzR9j0

Python Threading Tutorial by Corey Schafer
https://www.youtube.com/watch?v=IEEhzQoKtQU

Python Tutorials : Threading Beginners by PyMoondra
https://www.youtube.com/watch?v=bnm5_GH04fM

https://github.com/Richard-Burd/python-3-sandbox

last updated @ 11:14pm on 29/June/2021 by Richard Burd   rick.a.burd@gmail.com

About

Python Illustrated