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 /~/python3playlist $ python -V
// => This is the default version of Python installed, make sure you use a capital 'V', not 'v'

user@ubuntu /~/python3playlist $ python
// => This starts the python shell where you can type in Python code directly (press "Ctrl+D" to exit the shell)

user@ubuntu /~/python3playlist $ python3
// => This starts the Python shell but since Ubuntu uses Python 2.7 you have to have 2.7 installed

user@ubuntu /~/python3playlist $ 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 considered 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(5)#<class 'int'>
#=> integer

type(5.0)#<class 'float'>
#=> floats (or floating point numbers)

type('Hey')#<class 'str'>
#=> strings
```

- *integer — whole numbers*
- *floats (or floating point numbers) — anything with a decimal in them; needs to run through a python interpreter; it attempts the result of division*

Pythonistas call this an "argument"

```
10 % 3
# 1
```

division remainder or "modulus"

```
5 / 2#=> 2.5
5 // 2#=> 2
```

unless you use the double forward slash

BIDMAS stacking order

Brackets
Indicies
Multiplication
Addition
Subtraction

```
(5 + 2) * 3
# 21

2 ** 3 * 5 + 2
# 42

age = age + 1
# age => 10

age += 1
# age => 10

age = 5
age *= 2
# age => 10
```

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[2:]
#=> 'llo!'

greet[2:5]
#=> 'llo'
```

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

slicing

```
greet = 'Hello!'
name = 'Jim'
#=> 'Jim'

greet + name
#=> 'Hello!Jim'

name * 3
#=> 'JimJimJim'

name.upper()
#=> 'JIM'

name.lower()
#=> 'jim'

len(name)
#=> 3
```

.upper()

.lower()

add (concatenate) & multiply strings

Python has special methods for strings

```
cheese = 'Brie, Cheddar, Stilton'
cheeses = cheese.split(', ')
#=> ['Brie', 'Cheddar', 'Stilton']
```

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

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

.split()

.append()

```
list1 = ['Bob', 'Tom', 'Ann']
#=> ['Bob', 'Tom', 'Ann']

list1 + list ['Carl']
#=> ['Bob', 'Tom', 'Ann', 'Carl']

list1.append('Dan')
#=> ['Bob', 'Tom', 'Ann', 'Dan']

list1.pop()
#=> 'Dan'

list1 = ['Bob', 'Tom', 'Ann', 'Carl']

list1.remove('Tom')
#=> 'Tom'

del(list1[2])
#=> 'Ann'

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

next[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 and returns that object
.append() returns the list 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	Comprehension Method
`numbers = [5, 10, 15, 20, 100, 1000]`	`numbers = [5, 10, 15, 20, 100, 1000]`
`double_numbers = []`	`double_numbers = [number*2 for number in numbers]`
`for number in numbers:`	
`double_numbers.append(number*2)`	
`print(double_numbers)`	`print(double_numbers)`
`#=> [10, 20, 30, 40, 200, 2000]`	`#=> [10, 20, 30, 40, 200, 2000]`

same output

Another example below squares all even numbers from 1 to 10

Standard Method	Comprehension Method
`nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`	`nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`
`squared_even_numbers = []`	`squared_even_numbers = [`
`for num in nums:`	`num**2 for num in nums if(num%2 == 0)`
`if(num % 2 == 0):`	`]`
`squared_even_numbers.append(num**2)`	
`print(squared_even_numbers)`	`print(squared_even_numbers)`
`#=> [4, 16, 36, 64, 100]`	`#=> [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/timflies/list_&_tuple_manipulation.py

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

```
my_list = ['a', 'b', 'c', 'd', 'e']
#=> ['a', 'b', 'c', 'd', 'e']

my_tuple = ('a', 'b', 'c', 'd', 'e')
#=> ('a', 'b', 'c', 'd', 'e')

print(my_tuple[slice(1, 3)])
#=> ('b', 'c')

print(my_list[slice(1, 3)])
#=> ['b', 'c']

my_list.append('x')
#=> ['a', 'b', 'c', 'd', 'e', 'x']

my_list[-1] = 'z'
#=> ['a', 'b', 'c', 'd', 'e', 'z']

print(my_list)
#=> ['a', 'b', 'c', 'd', 'e', '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 func_decorator(function):
    print("Before calling " + function.__name__)
    function()
    print("After calling " + function.__name__)
    return function_wrapper

@func_decorator
def question():
    print("Can you give me a discount for that?")

question()
#=> "Before calling question"
#=> "Can you give me a discount for that??"
#=> "After calling question"
```

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

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*

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

if age < 18:
    print('you are under 18')
elif age >= 18 and age <= 21:
    print('You're a teenager.')
elif age >= 22 and age <= 35:
    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*

```
mainjam = ['rye', 'dry', 'ma', 'bob', 'ian']

for ninja in mainjam:
    if len(ninja) <= 2:
        # no an here...
        continue
    print('Hi ' + ninja)
#=> "Hi rye"
#=> "Hi dry"
#=> "Hi bob"

for ninja in mainjam:
    if ninja == 'ian':
        print(f'{ninja} - is the last ninja')
        break
    print('Hi ' + ninja)
#=> "Hi rye"
#=> "Hi dry"
#=> "Hi bob"
#=> "ian - is the last ninja"
```

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

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

```
age = 25
num = 0

while num < age:
    print("Hi ninja")
    print("Don's ninja" if (num % 2 == 0) else "Ian's ninja")
    num += 1
    if num == 10:
        continue
    print("I am ninja", num)
    print(num)
    num += 1
```

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

this will give us only even numbers

without these specific indentations, the Python code will not work

File Importing

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

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

python-3-sandbox/test.py

```
from lessons.space.planet import Planet

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

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

Console Inputs & Strings

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

python3 myScript.py

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

area = 3.142 * 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 /~/python3playlist $ 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 ····· type casting ····· print() ····· int()

But now let's look at some string formatting

```
num1 = 3.1415
num2 = 10.2893

print("The first variable is {0:f} and the second one is {1:f}".format(num1, num2))
#=> The first variable is 3.141500 and the second one is 10.289300

print("The first variable is {0:5} and the second one is {1:5}".format(num1, num2))
#=> The first variable is 3.141 and the second one is 10.289

print("The first variable is {0:10.2} and the second one is {1:10.2}".format(num1, num2))
#=> The first variable is 3.1415 and the second one is 10.2893

print("The first variable is {0:0} and the second one is {1:0}".format(num1, num2))
#=> The first variable is 3 and the second one is 10

name = input("Enter your name:")
print("Hey" + " " + my_name.capitalize() + "!")
#=> Hey Jim!

age = int(input("Enter your age:"))
print("Hey Jim! Your dog is " + str(age) + " years old!")
#=> Hey Jim! Your dog is 4 years old!
```

.format()

0 in the first ····· 1 in the set ····· list = [num1, num2]

strings

We could specify 10 or even 15 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 decimal number we want to see ····· maxlen

Dictionaries

Python dictionaries are similiar to Ruby hashes or JavaScript objects

```
person = {'name': 'Mario', 'occupation': 'Plumber', 'age': 35}

print('name' in person)
#=> True

print(person['name'])
#=> 'Mario'

print(person.get('name'))
#=> 'Mario'
```

see if key exists in the dictionary with `in`

find the value of a specified key

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

```
print(person.keys())
#=> dict_keys(['name', 'occupation', 'age'])

print(list(person.keys()))
#=> ['name', 'occupation', 'age']

print(list(person.values))
#=> ['Mario', 'Plumber', 35]

print(list(person.items()))
#=> [('name', 'Mario'), ('occupation', 'Plumber'), ('age', 35)]

my_name = input("Enter your name:")
print("Hey" + " " + my_name.capitalize() + ", you in the list!")
#=> Hey Jim, I saw you in the list!

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

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

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

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

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 ····· random

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

people_space = ['jim', 'bob', 'tim', 'tim']
type(people_space)
#=> <class 'list'>

my_var = 10
type(my_var)
#=> <class 'int'>

num_var = [1, 2, 3]
type(num_var)
#=> <class 'list'>

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

list

the `list()` *method takes a string and makes each character its own string in a new list like:* `['t', 'o', 'n']`

join

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

the init function

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

```
class Planet:
    def __init__(self, name, radius, gravity):
        self.name = name
        self.radius = radius
        self.gravity = gravity
        self.system = 'Sith System'

mustafar = Planet('Mustafar', 12, 9.5)
print(mustafar.name)
#=> 'Mustafar'
print(mustafar.radius)
#=> 12
print(mustafar.gravity)
#=> 9.5
print(mustafar.system)
#=> 'Sith 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

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

These are the instance attributes

```
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 distance(*, ly=None):
        if ly:
        return f'{ly} lightyears is {ly * 9.461:.3f} trillion kilometers'
        else:
        print("That's far away!")

    @classmethod
    def from_data(cls, name, radius, gravity, system):
        return cls(name, radius, gravity, system)

    def __str__(self):
        return f'{self.name}'

    def __repr__(self):
        return f'Planet({self.name}, {self.radius}, {self.gravity}, {self.system})'
```

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

static method

cls

class method

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

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

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*

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

python-3-sandbox/lessons/space/__pycache__

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

`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.planet.planet import planet, planet_wal

radius = Planet('Mabor', 300000.5, 'Mabor System')
radius_mass = planet_mass(radius, gravity, radius.radius)
radius_wal = planet_wal(radius.radius)

print(f'radius mass (mabor kg) = {radius_mass} and a volume of {radius_wal}')
#=> Mabor has a mass of 1.7946024968587e+22 and a volume of 377040000000.0

print(f'default: {radius.orbit()}')
#=> default: the planet mabor orbit at 2000 miles per hour

print(f'custom: {radius.orbit(250 mph)}') same as: print(f'custom: {Planet.orbit(250 mph)}')
#=> custom: the planet mabor orbit mphs at 250 mph

print(radius.__repr__())
#=> All planets are stored because of gravity

print(radius.__str__())
#=> Mabor

print(f'{radius.__dir__()}')
#=> ['__repr__', '__str__', '__format__', '__dir__', '__sizeof__', '__hash__', '__reduce__', '__reduce_ex__', '__doc__', '__eq__', '__ne__', '__lt__', '__gt__', '__le__', '__ge__', '__init__', '__delattr__', '__setattr__', '__getattribute__', '__subclasshook__', '__new__', '__class__', 'orbit', 'distance', 'from_data', '__str__', '__repr__', 'name', 'radius', 'gravity', 'system']
```

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

```
def planet_mass(gravity, radius):
    mass = (gravity*radius**2) / (6.67430*10**-11)
    return mass

def planet_wal(radius):
    wal = (4.19*(radius**3)) / 3
    return wal
```

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

this is a module

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

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

```
from functools import reduce
words = ['jim', 'tom', 'jim']
text = input("Username: ")

sentence = []

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

''.join(['r', 'u', 'n'])
#=> 'run'

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

join

map (function, list, that being operated on)

Standard Method	Vanilla Map Method
`for word in words:`	`print(map(jumble, words))`
`anagrams.append(jumble(word))`	`#=> [map object at 0x7fb4a5d7f710]`
`print(anagrams)`	
`#=> ['jim', 'mto', 'mjp']`	

map

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	Working Map Method
`print([jumble(word) for word in words])`	`print(list(map(jumble, words)))`
`#=> ['mjp', 'mto', 'jim']`	`#=> ['mjp', 'mto', 'jim']`

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', 'D', 'F', 'A', 'B']

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

print(filter(remove_bad_grades, grades))
#=> <filter object at 0x7fb4a5d7f7c0>

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

Using the Filter Method

filter (testing function, list being operated on)

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

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

Using Comprehension

filter

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

for my_grade in grades:
    if my_grade == 'F' or my_grade == 'D':
        continue
    print("Keep " + my_grade)
#=> "Keep A"
#=> "Keep B"
#=> "Keep C"
#=> "Keep A"
#=> "Keep B"
```

Using a For Loop

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/timflies/collections_deque.py

Deque is a special type of list; it's faster than a list for adding items to the beginning and end of the list, but if you're going to want to randomically 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('d')
#=> deque(['h', 'e', 'l', 'l', 'o', 'd'])

d.appendleft('z')
#=> deque(['z', 'h', 'e', 'l', 'l', 'o', 'd'])

d.extend('ello')
#=> deque(['z', 'h', 'e', 'l', 'l', 'o', 'd', 'e', 'l', 'l', 'o'])

d.extendleft('ello')
#=> deque(['o', 'l', 'l', 'e', 'z', 'h', 'e', 'l', 'l', 'o', 'd', 'e', 'l', 'l', 'o'])

d.pop()
#=> deque(['o', 'l', 'l', 'e', 'z', 'h', 'e', 'l', 'l', 'o', 'd', 'e', 'l', 'l'])

d.popleft()
#=> deque(['l', 'l', 'e', 'z', 'h', 'e', 'l', 'l', 'o', 'd', 'e', 'l', 'l'])

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

print(d)
#=> deque([])
```

the `Deque` *must be imported*

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

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

this will destructively clear all contents of the deque

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

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

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

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

Try & Accept

all code shown in this section is available here:
github.com/Richard-Burd/python-3-sandbox/timflies/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("Hi, {number} is in fact a legit username!")
except:
    print("{text} is not a legit username, minus decimal (.)")
```

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 = "Guy", time = "12:30pm"):
    print("Hey {name}! It's {time} right now!")
    return

greet("Jim", "3:30pm")
#=> "Hey Jim! It's 3:30pm 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 overridden

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

def volume(radius, 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)

radius *area* *length* *volume*

https://repl.it/@Richard_Burd/Variable-Scopedmain.py

```
my_name = "Jim"

def parent_scope():
    my_name = "Bob"
    print("I am my_name in the parent scope.")
    def child_scope():
        my_name = "Kid"
        print("I am my_name in the child scope.")
    def get_scope():
        print("I am my_name in the get scope.")
    print("I am my_name in the get scope.")

    # call the child function:
    child_scope()

    # call the parent function:
    parent_scope()

#=> I am Bob in the parent scope.
#=> I am Kid in the child scope.
#=> I am Bob in the get 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/timflies/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()

d = Dog()
#=> TypeError: unsupported operand type(s) for +: 'Queue' and 'int'
```

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(5)
print(cls)
#=> <class __main__.make_class.<locals>.Cat>

x = 10

x = deque("interlochen")
print(x)
#=> deque(['i', 'n', 't', 'e', 'r', 'l', 'o', 'c', 'h', 'e', 'n'])

x = deque("interlochen", maxlen=5)
print(x)
#=> deque(['l', 'o', 'c', 'h', 'e'], maxlen=5)

x = deque([1, 2, 3])
print(x)
#=> deque([1, 2, 3])

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

x.extendleft([7, 8, 9])
#=> deque([9, 8, 7, 1, 2, 3, 4, 5, 6])
```

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*

Here we're calling the class method

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

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

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/timflies/nested_function_calls.py

```
def some_function():
    def inner_function():
        print("get rekt with '*'")
    return inner_function

table = some_function()
table.__doc__ = "Flips table"

print(id(table))
#=> 139639020174832

print(type(table))
#=> <class 'function'>

print(inspect.getsource(table))
#=> def some_function():
#=>     def inner_function:
#=>         print("get rekt with '*'")
#=>     return inner_function

print(inspect.getsource(table))
#=> def inner_function:
#=>     print("get rekt with '*'")

print(inspect.getsource(table))
#=> OSERROR: could not find class definition
```

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

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

inspect

getsource

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

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': 'Tom',
        'work-history': [
            {'job-1': 'server',
            'job-2': 'bartender'},
        ],
        'bio': 'I was a server and a bartender'
    },
    'user-2': {
        'name': 'Tom',
        'work-history': [
            {'job-1': 'plumber',
            'job-2': 'doctor'}
        ],
        'bio': 'I was a plumber and a doctor'
    }
}

print(users['user-1']['work-history'][0]['job-1'])
#=> 'server'

print(users['user-1']['work-history'][0]['job-2'])
#=> 'bartender'

print(users['user-2']['work-history'][0]['job-1'])
#=> 'plumber'

print(users['user-2']['work-history'][0]['job-2'])
#=> 'doctor'

print(users['user-1']['bio'])
#=> 'I was a server and a bartender'

print(users['user-2']['bio'])
#=> 'I was a plumber and a doctor'
```

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

```
print(set(people_space.values()))
#=> {'Jim', 'Bob', 'Tim'}

print(len(people_space.keys()))
#=> 3

my_new_set = set(people_space)
print(my_new_set)
#=> {'people_space', 'name', 'age'}

print(type(my_new_set))
#=> <class 'set'>

print(people_space['name'])
#=> 'Jim'

print(people_space['age'])
#=> 'Bob'

print(people_space['people_space'])
#=> 'Tim'

print(people_space['name'])
#=> 'Jim'

print(people_space['age'])
#=> 'Bob'

print(people_space['people_space'])
#=> 'Tim'

print(people_space['name'])
#=> 'Jim'

print(people_space['age'])
#=> 'Bob'

print(people_space['people_space'])
#=> 'Tim'
```

The `values()` *method can be used to avoid using the key, value in items.items()* *pattern in the very top example·*
the `.keys()` *method can be used to find specific key values as well*

```
def work_history_profession_finder(users, profession):
    for new_key, new_value in users.items():
        for category_key, category_value in new_value.items():
            if category_key == 'work-history':
                for job_value in category_value:
                    for job_key, job_value in job_value.items():
                        if profession == job_value:
                            print(f'{new_value.get('name')} was a {profession}')
bio_profession_finder(users, 'doctor')
#=> 'Tom'

def work_history_profession_finder(users, profession):
    for new_value in users.values():
        for work_value in new_value['work-history'].values():
            if work_value == profession:
                print(f'{new_value.get('name')} was a {profession}')
work_history_profession_finder(users, 'doctor')
#=> 'Tom'
```

for loops can be used to find values within nested dictionaries

and in

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

keys values

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

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/lessons/collections.py

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

x = Counter('yellow')
print(x)
#=> Counter({'l': 2, 'y': 1, 'e': 1, 'o': 1, 'w': 1})

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

Point = namedtuple('Point', 'x y z time')
mark = Point(1, 2, 3, '12:30pm')
print(mark)
#=> Point(x=1, y=2, z=3, time='12:30pm')

x = OrderedDict('yellow')
print(x)
#=> OrderedDict([('y', 1), ('e', 1), ('l', 2), ('o', 1), ('w', 1)])

x = defaultDict(int, 'yellow')
print(x)
#=> defaultDict(int, {'y': 1, 'e': 1, 'l': 2, 'o': 1, 'w': 1})
```

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/timflies/collections_counter.py

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

```
from collections import Counter

x = Counter('yellow')
print(x)
#=> Counter({'l': 2, 'y': 1, 'e': 1, 'o': 1, 'w': 1})

x = Counter('l', 'y', 'l', 'e', 'o', 'w')
print(x)
#=> Counter({'l': 2, 'y': 1, 'e': 1, 'o': 1, 'w': 1})

x = Counter(lower=1, dogs=2)
print(x)
#=> Counter({'dogs': 2, 'lower': 1})

print(list(x.elements()))
#=> ['lower', 'dogs', 'dogs']

x = Counter('l', 'y', 'l', 'e', 'o', 'w')
print(list(x.elements()))
#=> ['l', 'l', 'y', 'e', 'o', 'w']

x = Counter('l', 'y', 'l', 'e', 'o', 'w', 'l', 'y', 'l', 'e', 'o', 'w')
print(list(x.elements()))
#=> ['l', 'l', 'l', 'l', 'y', 'y', 'e', 'e', 'o', 'o', 'w', 'w']

print("The two most common key·value pairs are {x.most_common(2)}")
#=> The two most common key·value pairs are [('l', 4), ('y', 2)]

x.subtract('l')
print(x)
#=> Counter({'l': 3, 'y': 2, 'e': 1, 'o': 1, 'w': 1})

x.subtract('l', 'y')
print(x)
#=> Counter({'l': 1, 'y': 1, 'e': 1, 'o': 1, 'w': 1})

print(x.keys())
#=> dict_keys(['l', 'y', 'e', 'o', 'w'])

print(x.values())
#=> dict_values([1, 1, 1, 1, 1])

print(x.items())
#=> dict_items([('l', 1), ('y', 1), ('e', 1), ('o', 1), ('w', 1)])

print(x * x)
#=> Counter({'l': 1, 'y': 1, 'e': 1, 'o': 1, 'w': 1})

print(x + x)
#=> Counter({'l': 2, 'y': 2, 'e': 2, 'o': 2, 'w': 2})

print(x - x)
#=> Counter({})

print(x & x)
#=> Counter({'l': 1, 'y': 1, 'e': 1, 'o': 1, 'w': 1})

print(x | x)
#=> Counter({'l': 1, 'y': 1, 'e': 1, 'o': 1, 'w': 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 like these two*

`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)

Collections - NamedTuple

all code shown in this section is available here:
github.com/Richard-Burd/python-3-sandbox/timflies/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')
mark = Point(1, 2, 3, '12:30pm')
print(mark)
#=> Point(x=1, y=2, z=3, time='12:30pm')

print(mark.x, mark.y, mark.z, mark.time)
#=> 1, 2, 3, '12:30pm'

class Person(namedtuple('Person', 'name age')):
    pass

def __repr__(self):
    return f'Person(name={self.name}, age={self.age})'

def __str__(self, x):
    if len(name) > 4:
        raise Exception("invalid argument, must be an integer")
    return f'My Personal{self.name}'

self.name = str + 2*" " + self.name + str

print(Person('Jim', 27))
#=> Person(name='Jim', age=27)

print(Person('Jim', 27).__str__('Jim'))
#=> 'My Personal Jim'

print(Person('Jim', 27).__str__('Jim'))
#=> Traceback (most recent call last):
#=> Exception: invalid argument, must be an integer
```

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*

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

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

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

_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*

_replace()

_asdict()

```
print(mark._asdict())
#=> OrderedDict([('x', 1), ('y', 2), ('z', 3), ('time', '12:30pm')]

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

my_new_mark = mark._replace(z=100)
print(my_new_value)
#=> Point(x=1, y=2, z=100, time='12:30pm')

gd = Data._make([1, 2, 3, 4])
print(gd)
#=> Data(x=1, y=2, z=3, time=4)
```

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*

_make()

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(1)
q.put(2)
q.put('three')
q.put(True)

print(q.qsize())
#=> 4

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

while not q.empty():
    print(q.get(), end = ' ')
#=> 1 2 three True
```

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 7 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)

```
q = queue.LifoQueue()

q.put('Bob')
q.put('Tom')
q.put('Jim')

while not q.empty():
    print(q.get(), end = ' ')
#=> Jim Tom Bob
```

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*

LifoQueue

Priority Queue (Integers)

```
import time

q = queue.PriorityQueue()

q.put(5)
q.put(4)
q.put(3)
q.put(2)
q.put(1)

for i in
```