PYTHON, CHAPTER 0

AHMED J. ZEROUALI

Abstract. The very basics of Python. The notes for more advanced topics (data structures and algorithms, machine learning libraries, reinforcement learning, time series analysis etc.) will be written directly in Jupyter notebooks.

01/24/22

Contents

1. Procedural programming in Python

1.1. **Variable types and basic manipulations.**

1.1.1. *Integers.*

- As in C++, integers are referred to as "int".
- There's a modulo operator called using "%".
- In Python, the power operator is "**".

```
>>> 8 % 3
2
>>> 2**5
32
>>>
```

1.1.2. *Variable assignments.*

- Python uses Dynamic Typing, i.e. we can use a variable as an int and then later as an array of strings.
- Useful because saves code-writing time, but can easily lead to bugs.
- To check the type of a variable, use "type()". For instance:

```
>>> var = 2
>>> type(var)
<class 'int'>
>>> x = 2.01
>>> type(x)
<class 'float'>
>>> sentence = "Dynamic Typing is weird..."
>>> type(sentence)
<class 'str'>
```

Remark 1.1. I'm writing these in the command prompt of Python.exe. To clear the screen:

```
import os; os.system("cls") # Use "cls" for Windows, "clear" for Linux.
```

Next, to abort the execution of a script, one usually finds the combinations "Ctrl +C", "Ctrl +D" or "Ctrl +Z" online. For some reason, these do not work on my current machine, and the way to abort is "Ctrl+Pause/Break".

1.1.3. *Strings.*

- Strings are arrays of characters. Using the operator "[begin:end:step]", one can extract substrings of a given string. The first character has position 0, not 1.

```
>>> sentence = "Dynamic Typing is weird..."
>>> type(sentence[0])
<class 'str'>
>>> sentence[6]
'c'
>>> sentence[0:6]
'Dynami'
>>> sentence[0:7]
'Dynamic'
>>> sentence[::2]
'DnmcTpn swid.'
```

- One can also use reverse indexing:

```
>>> sentence[-5]
'r'
>>> sentence[-8:-1]
'weird..'
>>>
```

- In Python, the length function for strings is "len()":

```
>>> len(sentence)
26
```

- Line breaks are done using the character "\n". Concatenations can be done using "+", and (somehow) there's also a multiplication of strings using "*".

```
>>> string_1 = "hello"
>>> string_2 = "world"
>>> string_3 = string_1 + " " + string_2
>>> string_4 = string_1 + "\n" + string_2
>>> string_5 = 3*(string_3+" ")
>>> print(string_3)
hello world
>>> print(string_4)
hello
world
>>> print(string_5)
hello world hello world hello world
>>>
```

- The string class comes with several methods, called using "myString.method(param)". To see these methods in notebooks, one types tab.

```
>>> string_3.upper() # .upper() for all upper caps, .lower() for all lower caps.
'HELLO WORLD'
>>> string_3.split() # Splitting at the space character by default, creates a list of strings
['hello', 'world']
>>> string_3.split("o") # Split at the character "o"
['hell', ' w', 'rld']
>>>
```

1.1.4. *Print formatting.* I'm skipping the notes of this part for now. These are sections 19 and 20 of Portilla's Python bootcamp. A notebook for examples is here:

- https://github.com/Pierian-Data/Complete-Python-3-Bootcamp/blob/master/00-Python%20Object%20and%20Data%20Structure%20Basics/03-Print%20Formatting%20with%20Strings.ipynb

1.1.5. *Lists.*

- Lists in Python are arrays of elements that could be of distinct types. Many operations (indexing, slicing, concatenation, len()) pertaining to strings can be done on lists.
- Some examples:

```
>>> varString = "This is a string"
>>> varFloat = 3.14159265359
>>> varInt = 19
>>> myList = [varString, varFloat, varInt]
>>> print(myList) ['This is a string', 3.14159265359, 19]
>>> myList[0] = "Lists are mutable"
>>> print(myList) ['Lists are mutable', 3.14159265359, 19]
>>> list_2 = myList + ["concatenating lists", 43, 8.1]
>>> print(list_2)
['Lists are mutable', 3.14159265359, 19, 'concatenating lists', 43, 8.1]
>>> myList.append(["use append also", 9])
>>> print(myList) ['Lists are mutable', 3.14159265359, 19, 'can also use append', 8, ['use append also', 9]] # .append() seems tricky to use. Returns None.
```

- Some useful methods here are .sort() and .reverse(). These return the "None" value

```
>>> stringList_0 = ["not", "in", "alphabetical", "order"]; intList_0 = [19, -3, 15, 31, 23]
>>> stringList_1 = stringList_0; intList_1 = intList_0
>>> stringList_1.sort(); print(stringList_1)
['alphabetical', 'in', 'not', 'order'] # If contains capital 1st letters, these are > lowercaps
>>> intList_1.sort(); intList_1.reverse(); print(intList_1)
[31, 23, 19, 15, -3]
>>>
```

1.1.6. *Dictionaries.*

- Dictionaries are non-ordered "arrays" of key-value pairs (they call it a "mapping" in the set-theoretic sense). One can access the contents using the keys, but we cannot perform the indexing/slicing done on lists.
- For now, I will just refer to the notebook:
  https://github.com/Pierian-Data/Complete-Python-3-Bootcamp/blob/master/00-Python%20Object%20and%20Data%20Structure%20Basics/05-Dictionaries.ipynb

1.1.7. *Tuples.*

- Tuples are very similar to lists, except that: (a) They are immutable; (b) They have many less methods available.
- Portilla explains that these aren't used very much in practice (for beginners at least), and that they mostly become relevant to ensure data/type integrity in larger programs.

- Again, I will just refer to Portilla's notebook for now:
  https://github.com/Pierian-Data/Complete-Python-3-Bootcamp/blob/master/00-Python%20Object%20and%20Data%20Structure%20Basics/06-Tuples.ipynb

1.1.8. *Sets.*

- Sets are another variation of lists. Unlike tuples however, they admit only one representative of each element.
- Used much later in Portilla's Python Bootcamp. Notebook:
  https://github.com/Pierian-Data/Complete-Python-3-Bootcamp/blob/master/00-Python%20Object%20and%20Data%20Structure%20Basics/07-Sets%20and%20Booleans.ipynb

1.1.9. *Booleans.*

- These are exactly what we would expect: variables taking either "True" or "False" as values.
- Examples:

```
>>> x = (8 % 3)
>>> y = (x == 2)
>>> z = (x == 0)
>>> y
True
>>> z
False
>>> -1 >2
False
>>> -1 >-2
True
>>>
```

1.2. **Input/Output with text files.** As with other languages, managing files has a pretty involved syntax. Lecture 28 of Portilla's course covers the following:

- **Opening and closing a file:** Done using the "open('fileName.extension')" function, which can be assigned to a variable. Two types of common mistakes occur here: (a) Either the filename is wrong, and Python return errno2; (b) Either the path of the file is wrong. More on this below. To close the file, one uses the .close() method.
- **Reading a file:** The .read(), .seek(0), and .readlines() methods.
- **File paths:** The double slashes for Windows, MacOS/Linux format of addresses.
- **The 'with' statement:** Allows to open a file with various permissions (read/write). One doesn't need to close the file when using "with", and this method is also useful for exceptions handling.
- **Read/write permissions in Python:** The second parameter of "open(file_name, mode='x')" is the "mode", taking values 'r' (read), 'w' (write), 'a' (append), 'r+' (read and write) and 'w+' (overwrites existing or creates new). The 'w' permission over-writes the files, while the 'a' permission adds lines to the *end* of the file.
- **Writing to a file:** Done using the .write(variable) method.

Let's look at concrete examples now. We'll work with an example file "IO_Example_0.txt" with 5 lines, each saying "Example text file line xx".

- The first way to open a file is to not use the "with" statement.

```
>>> dirAddress = 'C:\\Folder_Containing_IO_Example_0.txt'
>>> txtFileName = 'IO_Example_0.txt'
>>> myFile = open(dirAddress+'\\'+txtFileName, 'r')
```

- The .read() method prints out all the contents in one long string. The .readlines() method creates a list, each entry being one of the lines of the file. To reset the file reading pointer at the beginning, one used fileVar.seek(0).

```
>>> myFile.read()
'Example text file line 01\nExample text file line 02\nExample text file line 03\nExample
text file line 04\nExample text file line 05\n'
>>> myFile.read()
' ' # Returns "None", because EOF
>>> myFile.seek(0) # Sets pointer back to beginning of file
0
>>> myFile.readlines() # Returns a list
['Example text file line 01\n', 'Example text file line 02\n', 'Example text file line 03\n',
'Example text file line 04\n', 'Example text file line 05\n']
>>> myFile.close() # Always close after using a file without the "with" statement
```

- The second way to open a file uses the "with" statement, which creates a new block of code that has to be indented. In this case, we don't need to close the file, but the instructions in the new block need to match the read/write/append mode used at the very beginning. We'll also use the .write() method:

```
>>> with open(dirAddress+'\\IO_Example_1.txt', 'w+') as myFile: # Write mode
...     myFile.write('Example of .write() in Python') # The indentation is very important here.
...     myFile.write('\n Will also work with csv and avi files for instance') # Add a second line
...
29
52
>>> with open(dirAddress+'\\IO_Example_1.txt','r') as myFile: # Read mode
...     contents = myFile.readlines() # Create a new variable to keep file contents outside
...                                   # current block
>>> print(contents)
['Example of .write() in Python\n', ' Will also work with csv and avi files for instance']
>>>
```

1.3. **Comparison operators and Boolean composition.**

- The comparison operators are "==", "!=", ">", ">=", "<" and "<=".
- To compose Boolean variables, one uses the statements "and", "or" or "not".

1.4. **Python statements.** This is section 5 of Portilla's Python bootcamp.

1.4.1. *If, elif and else.*

- The basic syntax for the if/elif/else statements is as follows:

```
if condition_1:
    # Set of instructions when condition_1=true.
elif condition_2:
    # Set of instructions when condition_2=true.
...
else:
    # Set of instructions when condition_n all false.
```

  Note the presence of the ":" symbol replacing the C++ "{" bracket, and keep in mind that the indentation is crucial to Python. There is no "end" statement here (as in Matlab), and it is the indentation that that gives the end instruction (as opposed to the "}" bracket in C++).
- Here's a concrete example.

```
loc = "Office"
if (loc == 'Home'):
    print('I am at home. How can I help you?')
elif (loc == 'Office'):
    print('I am at currently at the office. Could you call me back?')
elif (loc == 'Store'):
    print('I am doing some shopping right now.')
else:
    print('I am out right now. What\'s up?')
>>> I am at currently at the office. Could you call me back?
```

1.4.2. *For loops.*

- The syntax of a for loop is different from what we are used to.

```
for iter_var in iterable:
    # Instructions for each value of iter_var
```

  Note again the presence of colons before the indentation.
- In the syntax above, there is a lot of flexibiliy on the "iterable"'s class: It need not be an array of integers as in Matlab or C++, it could be a list, a string, a tuple, a dictionary, a range(m,M-1) etc.
- This example will print the first 8 terms of the Fibonacci sequence:

```
for i in range(0,7+1):
    if i==0:
        F_i, F_im1, F_im2 = 0,0,0
    elif i==1:
        F_i =1
    else:
        F_i = F_im1+F_im2
    print('F_',i,' = ',F_i)
    F_im2 , F_im1 = F_im1, F_i
```

- The next loop prints every character of "Hello world!" on a new line:

```
for i in 'Hello world!':
    print(i)
```

1.4.3. *While loops.*

- As usual, while loops are defined in terms of a Boolean condition.

```
while Boolean_Condition:
    # Instructions
else:
    # Instructions before stopping loop
```

1.4.4. *Addendas.* Some notes on addendas in Portilla's Python Bootcamp

- Sec. 5, Lecture 36: Introduces the statements "break", "continue" and "pass" for loops at the end.
- Sec. 5, Lecture 37: The functions range(); enumerate(); zip(); the "in" keyword; the min()/max() functions for elements of a list; the built-in "random" library.
- Sec. 5, Lecture 37: The input('Message') function for user input.
  - The argument of this function is the message displayed to the user, and input() always takes in a string.
  - To convert the user input to an integer or a float one uses int(input_string) or float(input_string) resp.
  - **Note (22/01/19):** More notes for the previous 3 items could be useful. Found enumerate() and reversed() in Lerner's book, could be relevant for interview questions. There also seems to be a bug in input() when called from an external file.
  - **Note from Sec. 7, Lect. 61:** Never execute a Jupyter cell containing input() twice. The kernel waits for an input after the first execution, the second execution just erases the prompt of the cell, but now no other cell can be executed. In this case the kernel has to be restarted and the previous code re-executed.
- Sec. 5, Lecture 38: Discusses List Comprehension, i.e. how to fill lists using loops in a much more compact form, and possibly use if/else conditions. Admittedly, makes the code harder to understand. There's also the possibility to nest loops.

```
# Square numbers in range and turn into list
>>> lst = [x**2 for x in range(0,11)]
>>> print(lst)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
# Check for even numbers in a range
>>> lst = [x for x in range(11) if x % 2 == 0]
>>> print(lst)
[0, 2, 4, 6, 8, 10]
```

1.5. Methods and functions.

1.5.1. *Methods.*

- By definition, a method is a built-in function associated to a class, and is called using objVar.method(arguments). In notebooks, one can get the help pop-up by pressing the tab key, and in the console, one uses the help() function.

1.5.2. *Functions.* This part starts at Lecture 43 in Section 5 of the Python Bootcamp, and extends with exercises etc. to Lecture 58.

- The syntax for a function definition is as follows:

```
def function_name(input_variables):
    "'
       Docstring describing the function
    "'
    # Function instructions
    return output_variables
```

- By convention in Python, the names of functions are written in lower caps with underscores (snake casing).
- The triple apostrophes delimit the string describing the function. This is called using "help(function_name)".
- Note the return keyword at the end. It is not strictly necessary, although it's a good practice to add "return None" if the function doesn't return values. Note that Python exits the function block once it reaches "return" (no instructions below it are executed).
- Typically, if there are several output variables returned by the function, one uses a tuple containing these values.
- Unlike C/C++: Python does not require function prototypes; one doesn't need to specify the classes of input/output variables.
- **The "*args" input:** The "*" operator allows one to be flexible on the number of arguments passed to a function, without having to pre-define the number of arguments (inputs) for function calls.

```
>>> def pct5_sum(*args):
...         "' Returns 5% of the sum of arguments."'
...         return (sum(args)*0.05)
...
>>> pct5_sum(50,50)
5.0
>>> pct5_sum(10,20,30,40)
5.0
>>>
```

  Note that during runtime, "args" is stored as a tuple inside the function.

- **The "**kwargs" input:** The "**" does the same as "*", but here we use a dictionary instead of a tuple, hence the kw for "key-word" arguments. Here's an example: https://github.com/Pierian-Data/Complete-Python-3-Bootcamp/blob/master/03-Methods%20and%20Functions/07-args%20and%20kwargs.ipynb

```
def myfunc(*args, **kwargs):
    if 'fruit' and 'juice' in kwargs:
        print(f"I like {' and '.join(args)} and my favorite fruit is {kwargs['fruit']}")
        print(f"May I have some {kwargs['juice']} juice?")
    else:
        pass
myfunc('eggs','spam',fruit='cherries',juice='orange')
```

- The *args and **kwargs inputs are useful when dealing with libraries (**Reference?**).

1.5.3. *Map and filter, Lambda Expressions.* In Lecture 55, Portilla discusses the "map" and "filter" functions, as well as Lambda Expressions. The "map" function works as follows.

- Consider some function "my_func()", and that one wants to apply it to each element of an iterable "in_iter". This can be done using the map function with syntax:

```
map(my_func, in_iter)
```

- The first argument is the desired function, called without the "()" at the end, and the iterable could be a list, a tuple, a string etc. The "map" function returns a memory address.

```
>>> iter_tuple = (-1,-2,-3,-4,5,6,7)
>>> map_out=map(is_even,iter_tuple) # is_even returns True for even integer
<map object at 0x0000024EB227C610>
>>> print(list(map_out))
[False, True, False, True, False, True, False]
```

We used "list()" to store/view the outputs of each element of iter_tuple.

Next we look at the "filter" function

- Suppose now that the function "my_func()" returns a Boolean only, and "in_iter" is again an iterable. The "filter" function returns the elements of "in_iter" for which "my_func()" returns true. The syntax is as follows:

```
filter(my_func, in_iter)
```

and continuing with the previous prompt example:

```
>>> iter_tuple = (-1,-2,-3,-4,5,6,7)
>>> filter_out = filter(is_even, iter_tuple)
<filter at 0x24eb227c730>
>>> print(list(filter_out))
[-2, -4, 6]
```

- Note that both "map" and "filter" produce iterables. With the same function and tuple above:

```
>>> for i in map(is_even, iter_tuple):
>>> ...     print(i)
False
True
False
True
False
True
False
>>> for i in filter(is_even, iter_tuple):
>>>...     print(i)
-2
-4
6
```

Now onto Lambda Expressions, also called anonymous functions.

- Lambda Expressions are useful in cases where we won't need a fully-developed function more than once. It is also particularly useful when utilized in conjunction with "map" and "filter".
- The syntax is as follows:

```
lambda input_var: # Short set of instructions
```

For instance, the following lambda expression returns squares of "input_var":

```
>>> num_sq = lambda in_int: in_int**2
>>> num_sq(5)
25
```

- A more practical example is the following, where the "is_even()" example is replaced by a Lambda Expression:

```
>>> list(filter(lambda in_int: (in_int%2==0), iter_tuple))
[-2, -4, 6]
```

- For the sake of readability of the code, it's better to reserve Lambda Expressions for simple tasks and use regular functions otherwise.

1.5.4. *Namespace and scope.* This is lecture 56, which explains the hierarchy of variable names in Python.

- The first main point is the LEGB rule, which stands for:
  - L: Local - Names assigned in any way within a function (def or lambda), and not declared global in that function.
  - E: Enclosing function locals - Names in the local scope of any and all enclosing functions (def or lambda), from inner to outer.
  - G: Global (module) - Names assigned at the top-level of a module file, or declared global in a def within the file.
  - B: Built-in (Python) - Names preassigned in the built-in names module : open, range, SyntaxError,...
- The second point is that for a variable which is globally defined, but that one wants to change locally within a function, one uses the keyword "global" in the definition of the function.

For examples: https://github.com/Pierian-Data/Complete-Python-3-Bootcamp/blob/master/03-Methods%20and%20Functions/06-Nested%20Statements%20and%20Scope.ipynb

1.5.5. *Functions and decorators.* Discussed in Section 12, Lectures 98-99. This is optional for beginners, and it is used a lot in pratice in Web Development with Python.

- The idea of decorators is to add instructions to an already existing function, without actually re-writing the function from scratch and potentially losing old work. The idea is to be able to toggle new functionalities on and off.
- A first prerequisite here is that one can define functions inside functions in Python (but they remain only locally defined), functions can return functions, and one can also pass functions as arguments in this language.
- When using functions as variables however, one doesn't add the parentheses "()" at the end. These are only used for execution.

```
# Returning functions
def hello(name='Jose'):
    def greet():
        return '\t This is inside the greet() function'
    def welcome():
        return "\t This is inside the welcome() function"
    if name == 'Jose':
        return greet
    else:
        return welcome
# Functions as arguments
def hello():
    return 'Hi Jose!'
def other(func):
    print('Other code would go here')
    print(func())
----------------------------------------
>>> other(hello)
Other code would go here
Hi Jose!
```

- Using the facts above, here is a quick and dirty example of a decorated function. We want to decorate the function "factorial(int)":

```
def factorial(in_int):
     out_int=1
     if in_int ==0:
          out_int =1
     elif (in_int<0):
          out_int =0 print('Error: Argument is negative')
     else:
          for i in range(1,in_int+1):
               out_int = out_int*i
     return out_int
```

in such a way that it asks the user for an input, say by making "factorial2()" as follows:

```
def aug_factorial(fnc):
    def prmpt_fact():
        in_int = int(input('Enter a positive integer N:\n'))
        print(f'N! = {fnc(in_int)}')
    return prmpt_fact

factorial2 = aug_factorial(factorial)
```

- Instead of reassigning "factorial()" or making "factorial2()" as above, we could write "aug_factorial(fnc)", and then write:

```
@aug_factorial
def factorial(in_int):
    # Same instructions as previously
------------------------------
>>> factorial()
>>> Enter a positive integer N:
6
N! = 720
```

- To clarify why they're called "decorators", think of the original function as a present, and the additional instructions as decorations.
- Link to the Notebook: https://github.com/Pierian-Data/Complete-Python-3-Bootcamp/blob/master/10-Python%20Decorators/01-Decorators.ipynb

1.5.6. *Generator functions.* This is Section 13, Lectures 100-102. The main takeaway is the "yield" statement used instead of "return" in a generator. The two other keywords introduced in this lecture, "next" and "iter", are rarely used however.

- To explain a generator in practical terms, we could write the following function that stores the n first Fibonacci numbers in a list:

```
def fibonacci(n):
    a,b =1,1
    out_lst = []
    for i in range(n):
        out_lst.append(b)
        a,b = b, a+b
    return out_lst
```

If n is pretty large for instance, this implementation would not be effective in terms of memory management.

- What we could do instead is create a **generator**, which is a function that gives one value of the sequence instead of storing all of the values. For the Fibonacci sequence, we could write:

```
def fibonacci_gen(n):
    a,b =1,1
    for i in range(n):
        yield b
    a,b = b, a+b
------------------------
>>> g = fibonacci_gen(5)
```

Notice that "return" is replaced by "yield" in this implementation. Instead of completely exiting the function, the execution is paused after the "yield" statement.

- To call the next number generated by "fibonacci_gen(n)", we assign the latter to a variable "g", and use the keyword "next(g)":

```
>>> print(f'next(fibonacci_gen(5))={next(g)}')
next(fibonacci_gen(5))=1
>>> print(f'next(fibonacci_gen(5))={next(g)}')
next(fibonacci_gen(5))=2
>>> print(f'next(fibonacci_gen(5))={next(g)}')
next(fibonacci_gen(5))=3
>>> print(f'next(fibonacci_gen(5))={next(g)}')
next(fibonacci_gen(5))=5
>>> print(f'next(fibonacci_gen(5))={next(g)}')
next(fibonacci_gen(5))=8
>>>
```

After the 5th call to "next(g)" we get an error (the function stopped generating).

- The last notion introduced in this lecture is the "iter" keyword. Essentially, an "iterator" for Python is a type of generator over which we iterate (**Beware:** We used the term iterator for objects over which we could loop, it's not the same here). For instance, if "str_var" is a given string, the command "next(str_var)" will not yield a letter from said string, and would instead return "TypeError" since "str_var" isn't an iterable. One instead uses "next(iter_str_var)", where iter_str_var = iter(str_var).
- Link: https://github.com/Pierian-Data/Complete-Python-3-Bootcamp/blob/master/11-Python%20Generators/01-Iterators%20and%20Generators.ipynb

1.6. **Advanced objects and data structures.** This covers Section 20 of the course, Lectures 138 to 144.

1.7. **Useful Python modules.** This subsection is covered in Section 14 of Portilla's course (Lectures 103-115), entitled "Advanced Python modules". Essentially a series of complementary modules.

1.7.1. *Collections module.* The *collections* module provides specialized container datatypes, as alternatives to the usual ones (*list*, *tuple*, *dict* and *set*). From the general documentation at: https://docs.python.org/3/library/collections.html, we have the following classes:

namedtuple()	factory function for creating tuple subclasses with named fields
deque	list-like container with fast appends and pops on either end
ChainMap	dict-like class for creating a single view of multiple mappings
Counter	dict subclass for counting hashable objects
OrderedDict	dict subclass that remembers the order entries were added
defaultdict	dict subclass that calls a factory function to supply missing values
UserDict	wrapper around dictionary objects for easier dict subclassing
UserList	wrapper around list objects for easier list subclassing
UserString	wrapper around string objects for easier string subclassing

Let's illustrate with the containers covered in Lecture 104.

Counters - Counter():

- This is a class inherited from *dict*. Its general purpose is to count the number of elements in a container, which could be a list, a tuple or a string.

- When instantiating a *Counter()* object, a container is passed as a parameter, and the returned object is similar to a dictionary, where the keys are the elements of the container, and where the values are the object counts.
- Below are examples where the container is a list, a tuple, a string, or a sentence.

```
>>> import collections

# Counting elements in a list:
>>> collections.Counter([1,2,2,3,3,3,4,4,4,4])
Counter({1: 1, 2: 2, 3: 3, 4: 4})

# Counting elements in a tuple:
>>> collections.Counter((1,1,1,1,2,2,2,3,3,4))
Counter({1: 4, 2: 3, 3: 2, 4: 1})

# Counting characters in a string:
>>> collections.Counter("\'Tenet\'")
Counter({"'": 2, 'T': 1, 'e': 2, 'n': 1, 't': 1})

# Counting distinct words in a sentence:
>>>collections.Counter( ("I am tomorrow, I am the end".lower()).split())
Counter({'am': 2, 'end': 1, 'i': 2, 'the': 1, 'tomorrow,': 1})

# Convert to a dictionary:
>>> dict(collections.Counter("\'Tenet\'"))
{"'": 2, 'T': 1, 'e': 2, 'n': 1, 't': 1}
```

- Portilla gives a couple of common patterns when using a *Counter()* object in the notebook of Lecture 104:
https://github.com/Pierian-Data/Complete-Python-3-Bootcamp/blob/master/12-Advanced%20Python%20Modules/00-Collections-Module.ipynb

Default dictionaries - defaultdict():

- When manipulating usual dictionaries, one obtains an error when trying to access non-existent keys. The *defaultdict* class allows to create new keys with default values provided by a function.
- Portilla uses lambda expressions for the "factory" function.

```
# Create default dictionary. Set first 2 keys, and call a 3rd.
>>> def_dict = collections.defaultdict(lambda: "Unassigned")
...     def_dict["Key_1"] = 1
...     def_dict["Key_2"] = 2
...     def_dict["Key_3"]
...     print(dict(def_dict))
{'Key_1': 1, 'Key_2': 2, 'Key_3': 'Unassigned'}
```

Named tuples - namedtuple():

- There are 2 ways of thinking of named tuples. The first way is a tuple for which entries have a "name" on top of an integral index. The second way of thinking of them is as objects with attributes corresponding to the names.
- Here is one of Portilla's examples:

```
# Create the Dog namedlist and 2 examples
>>> Dog = collections.namedtuple("Dog", ["Breed", "Age", "Name"])
...     dog_1 = Dog(Breed = "Collie", Age = 3, Name = "Lassie") #
...     dog_2 = Dog(Breed = "Pitbull", Age = 2, Name = "Tyson")

# The attributes/values can be accessed via names or indices:
>>> dog_1[2]
'Lassie'
>>> dog_2.Name
'Tyson'
>>> dog_1.Breed
'Collie'
>>> dog_2[0]
'Pitbull'
```

1.7.2. *OS module, files and folders.* This part follows Lecture 105, covering the shell utilities module and the OS module. This lecture covers the basics of file manipulation, essentially just a couple of commands. The documentations can be found at the following links:

- OS: https://docs.python.org/3/library/os.html.
- Shutil: https://docs.python.org/3/library/shutil.html.
- Portilla's notebook: https://github.com/Pierian-Data/Complete-Python-3-Bootcamp/blob/master/12-Advanced%20Python%20Modules/01-Opening-and-Reading-Files-Folders.ipynb

For illustration, I'm using a Google Colab notebook online, which is why the file/directory paths follow the Unix format. The imports are:

```
>>> import os
>>> import shutil
```

Contents of a directory:

- ***os.listdir("dir_path")*** is the equivalent of *dir* in the command line. The argument is the path of the directory one wishes to explore, and this functions outputs a list of strings giving the contents of the folder. If no argument is specified, this function returns the contents of the pwd.
- Some examples:

```
# List contents of pwd (colab notebook)
>>> os.listdir()
['.config', '.ipynb_checkpoints', 'test_file.txt', 'sample_data']

# List contents of "sample_data"
# -> Pass directory path as arg. to os.listdir()
>>> os.listdir("sample_data")
['README.md',
'anscombe.json',
'california_housing_train.csv',
'mnist_train_small.csv',
'california_housing_test.csv',
'mnist_test.csv']
```

Moving files:

- This is cut and paste of any OS for files. Done using ***shutil.move("file_path", "target_dir")***:

```
# Moving files done with .move(),
# Need admin permissions...
>>> shutil.move("test_file.txt", "sample_data")
# Outputs new filepath:
'sample_data/test_file.txt'
```

Deleting files:

- There are 4 ways of deleting files in Lecture 105:
  (1) Using ***os.unlink("file_path")***, which deletes one file only.
  (2) Using ***os.rmdir("dir_path")***, which deletes only empty folders.
  (3) Using ***shutil.rmtree("dir_path")***, which deletes a folder and the entirety of its contents
  (4) Using ***send2trash("file_path")***, a safer method that sends files to recycle bin for possible later recovery. The previous methods are irreversible.
- Here's an example with *send2trash*:

```
>>> import send2trash
>>> send2trash.send2trash("sample_data/test_file.txt")
```

(I don't know where the recycle bin is in Colab).

***os.walk()*:**

- Using ***os.walk("dir_path")*** produces a generator *(folders, subfolders, files)* giving a tree representation of all contents of the argument directory.
- For an example, here is a verbatim script to explore the contents of the "usr/local/etc" folder in Colab:

```
# First we instantiate the iterable:
>>> w_usr = os.walk("../usr/local/etc")

# To explore the folder, loop over folders, subfolders and files:
>>> for folder, sub_folders, files in w_usr:
>>>...     print(f"Currently looking at {folder}")
>>>...     print("----------------\n") print("The folders are: ")
>>>...     for sub_fold in sub_folders:
>>>...         print(f"\t Subfolder: {sub_fold}")
>>>...         print("\n")
>>>...         print("the files are: ")
>>>...         for f in files:
>>>...             print(f"\t File: {f}")
>>>...             print("----------------\n")
```

1.7.3. *Datetime module.*

1.7.4. *Math and Random modules.*

1.7.5. *Debugger.*

1.7.6. *Regular expressions.*

1.7.7. *Timing code execution.*

1.7.8. *Compressing files.*

2. Object oriented programming in Python

2.1. **Fundamentals of classes and inheritance.** The material below is from Section 8 of Portilla's lectures (68-76).

2.1.1. *Basic syntax.* Declaring a new class in Python follows the syntax below:

```
class NameOfClass():

    # Class object attributes: Attributes shared by all instances of this class
    #                          Referred to using self.class_obj_attr_i
    class_obj_attr_1 = val_1
    class_obj_attr_2 = val_2

    # Constructor: Note double underscores of "init" and the "self" variable
    # Optional: When instantiated without "par_i" values,
    #           the cons'tor assigns "par_i_deflt" by default
    def __init__(self, par_1=par_1_deflt, par_2=par_2_deflt):
        # Class attributes defined here
        self.par_1 = par_1
        self.par_2 = par_2
        # etc. ...

    # Class methods: Note keyword "self" always passed as param

    ## Called using my_obj.method_1()
    def method_1(self):
        # Method instructions

    ## Called using my_obj.method_2(args)
    def method_2(self, some_args):
        # Method instructions
```

- By convention in Python, class names are written in "camel casing", i.e. no underscores and capitalization.
- The "self" keyword represents an instance of the class, and is an implicit variable in other object-oriented languages. In Python however, it needs to be explicitly passed as parameter, in order to connect the attributes/methods to the new class. The word "self" is more of a convention, the important thing is to have an explicit instance variable passed inside the class.
- The attributes of the class are defined in the "__init__()" block (for "initialize"). To create a new instance of a class, one uses parentheses: "new_obj = NameOfClass()" for default attributes, or "NameOfClass(par_1_val,par_2_val)".
- The syntax to call an attribute is "my_object.attribute" (no parentheses), and to call a method: "my_object.method(params)" (parentheses).
- Just to emphasize *class object attributes*: These are attributes shared by all instances of the class, which could be thought of as class "invariants". Class object attributes

are defined **above** the "def __init__" block, and below that, they're always referred to using the "self" keyword.
- Lectures 68-76 do not address the public/private/protected properties. For private members one prefixes the names with two underscores, and for protected members the prefix is one underscore.

2.1.2. *Inheritance and polymorphism.* This part is discussed in Lecture 71. The inheritance syntax in Python is much more accessible than that of C++. The lectures do not go very deep into the subtleties (e.g. constructors of derived class).

```
class BaseClass():
    def __init__(self):     # Base Constructor
        # Class attributes defined here
        self.par_1 = par_1
        self.par_2 = par_2
        # etc. ...
    def method_1(self):
        # Base Class method_1
    def method_2(self, some_args):
        # Base Class method_2

# Given the BaseClass above: Derived classes are defined as follows
class DerivedClass(BaseClass):
    # New Constructor.
    def __init__(self):
        BaseClass.__init__() # Not necessary
        self.par_1 = par_1
        self.par_2 = par_2
        # etc. ...
    # Methods of derived class
    def method_2(self, some_args):
        # Derived Class method_2. Overwrites the Base Class method.
    def method_3(self, some_args):
        # Derived Class method_3
```

- To add attributes to derived classes, one writes them in the new constructor. A detail that is unclear to me at the moment is the rôle of "BaseClass.__init__()" in the constructor of a derived class.
- To overwrite a method of the BaseClass, it suffices to use the same method in the derived classes and re-implement it.
- To make a class virtual, so that its methods are only implemented by derived classes, one can raise the following error:

```
raise NotImplementedError("Some error message")
```

- Polymorphism seems to work easily in Python. See notebook for concrete examples. Add more on this topic later?
- Link to notebook:

https://github.com/Pierian-Data/Complete-Python-3-Bootcamp/blob/master/05-Object%20Oriented%20Programming/01-Object%20Oriented%20Programming.ipynb

2.1.3. *Special methods (overloading).* This is in Lecture 72. Without going into a general discussion using other sources, I'll just summarize the given examples.

- The motivation examples that Portilla uses are those of "len()" and "print()", which respectively expect an integer and a string, and which we'd like to use on a new class. The function "len()" would just return an error, while "print()" will return the memory address of the object.
- To overload built-in Python functions, one adds a method with double underscores "__built-in-f'n__()" to the class definition.
- To overload "print()" for example, one implements a string representation of an instance of our class by adding the following method:

```
def __str__(self):
    return f'String describing object using {self.attribute}'
```

- Similarly, overloading "len()" is done by returning an integer attribute:

```
def __len__(self):
    return self.int_attribute
```

- One function that hasn't been discussed previously is "del", which erases variables from the memory. Since it's useful to print a deletion message for debugging, one can use the following:

```
def __del__(self):
    print(f'An object of class {type(self)} with {self.attribute} has been deleted')
```

- In Python jargon, these special methods are called "magic" or "dunder" methods (for double underscores).

2.2. **Modules and packages.**

2.2.1. *Summary.* This is covered in Lectures 77-79. Won't write too much stuff for now. The following topics are covered:

- **Lecture 77:** Using pip install in cmd to install Python packages. Portilla refers to the PyPI repository online to download/install and get the pip command for specific packages (pip comes installed with the Anaconda distribution, PyPI stands for "Python Package Index").
- **Lecture 78:** How to create one's own Python packages and file handling. Modules are *.py scripts called by other *.py scripts. A package is a collection of modules.
- Indicating that the *.py scripts in a folder constitute a package is done by adding a blank file to said folder. The empty file has to be **called "__init__.py"**.
- Say there's a function called "my_func" in a module called "myModule.py". The inclusion command is:

```
from myModule import my_func
```

- For large packages with several sub-packages, one creates a subfolder for each subpackage, **each of which** should contain a "__init__.py" script.
- In Lecture 78, Portilla focuses on importing modules, submodules and functions. The packages, subpackage and module names are separated by a ".", so that the syntax is:

```
# Module
import MainPackage.Module_in_Main_Package

# Function in module, called using:
# MainPackage.Module_In_Main_Package.func_in_module()
from MainPackage.Module_In_Main_Package import func_in_Module

# Submodule,i.e. module inside a subpackage:
import MainPackage.SubPackage.Module_in_Subpackage

# Function in submodule, called using:
# MainPackage.SubPackage.Module_in_Subpackage.func_in_SubModule()
from MainPackage.SubPackage.Module_in_Subpackage import func_in_SubModule
```

- Continuing with the previous example, it is useful in practice to rename the functions imported from (sub)modules for more readable code. This is done using the "as" keyword after "import":

```
# Function in module, called using func_in_module():
from MainPackage.Module_In_Main_Package import func_in_Module as func_in_Module

# Function in submodule, called using func_in_SubModule()
from MainPackage.SubPackage.Module_in_Subpackage import func_in_SubModule as func_in_SubModule
```

- **Lecture 79:** Discusses the built-in variables "__name__" and "__main__".
- If a module is executed directly, Python views essentially it as the "int main(void)" function of C++. In this case, we have __name__ = "__main__".
- If a module is being imported however, "__name__" isn't equal to "__main__", as the main function must be from another script.

Remark 2.1. A couple of new tricks I learned along the way.

- While working on my example of package, I came accross the analogs of cmd's "dir" and "cd" commands in Python. Again, these are from the "os" library.

```
>>> import os
>>> os.getcwd() # Returns current working directory
'C:\\Program Files\\Python\\Python39'
>>> os.chdir('C:\\Users\\Me\\Documents\\Python\\First steps\\ZAJTestPackage') # Analog of "cd" in cmd
>>> os.listdir() # Analog of "dir" in cmd
['TestCountPermute.py', 'TestSubPack1', 'TestSubPack2', '__init__.py', '__pycache__']
>>>
```

- Another useful feature I found is the possibility of running *.py scripts inside Jupyter notebooks, using:

```
%run myScript.py
```

I'm assuming that myScript is in the working directory of the notebook.

2.3. **Errors and exception handling.**

2.3.1. *Basics.* This is Lecture 80. The notebook is here:
https://github.com/Pierian-Data/Complete-Python-3-Bootcamp/blob/master/07-Errors%20and%20Exception%20Handling/01-Errors%20and%20Exceptions%20Handling.ipynb

The main thing to remember is that there are 3 keywords corresponding to blocks of instructions:

- **try:** Python attempts this block of code, which is fully executed if it doesn't lead to any errors.
- **except Error_Type:** If Error_Type occurs in the "try" block, this specific exception is raised. Note that one can make several of these blocks with the last one starting with "**else:**".
- **finally:** The final set of instructions, which is executed regardless of whether there's an error or not.
- Portilla gives the following example, where a function asks the user for a number, and raises an error whenever the input string can't be converted to an int.

```
def askint():
    try:
        val = int(input("Please enter an integer: "))
    except:
        print("Looks like you did not enter an integer!")
        val = int(input("Try again-Please enter an integer: "))
    finally:
        print("Finally, I executed!")
    print(val)
```

2.3.2. *Pylint.* The Pylint package checks scripts for styling, names of variables, warnings, errors etc. It's like the prompt at the bottom of Visual C++, which reports warnings and whatnot. To install pytlint, one runs the following in cmd:

```
pip install pylint
```

Next, one asks Pylint to produce a report on a script with the following commend in cmd:

```
pylint myScript.py
```

2.3.3. *Unit Test.* Unit Test is a built-in library of Python that allows one to make tests for functions. In Lecture 85, Portilla gives the example of a function "capitalize(text)" that should capitalize the first letter of each word in the string "text". The test script contains the following instructions:

```
import unittest
import cap

class TestCap(unittest.TestCase):
    def test_one_word(self):
        text = 'python'
        result = cap.cap_text(text)
        self.assertEqual(result, 'Python')

def test_multiple_words(self):
    text = 'monty python'
    result = cap.cap_text(text)
    self.assertEqual(result, 'Monty Python')

def test_with_apostrophes(self):
        text = "monty python's flying circus"
        result = cap.cap_text(text)
        self.assertEqual(result, "Monty Python's Flying Circus")


if __name__ == '__main__':
    unittest.main()
```

Might update this subsection if I learn more. The strange thing here is that the test cases are defined as inherited classes.

3. Advanced tasks with Python

3.1. **Overview (temp).** I am writing this on 21/06/14. Here are the sections remaining in Portilla's Python course at the moment:

12 Decorators. (More on functions)
13 Generators. (More on functions)
14 Advanced modules. Contains a lot of useful and interesting material, including: OS; Debugger; Timing code; Zipping...
15 Web Scraping.
16 Working with images.
17 Working with PDFs and CSVs.
18 Emails.
20 Advanced objects and Data Structures. (More on numbers, strings, sets dictionaries and lists).
21 Intro to Graphic User Interfaces (GUIs).

Sections 12, 13, 14 and 20 will be summarized in the "Procedural Programming" section of these notes. The present section of these notes will summarize the "actually important" tasks for practice, meaning sections 15-18 and section 21.

3.2. **Web scraping.** This is Section 15 of Portilla's Python course, Lectures 116-124.

- Web scraping is the automation of data gathering of website (text, images etc.). This type of task requires a basic understanding of how webpages are written, and how the front-end of a website works.
- A note on permissions for web scraping: Some websites block IP addresses when scraping software is detected. There is usually some limit on how many times a website admits visits and a section on permissions.
- One of the limitations of web scraping is that each website is unique, and the code should be modified according to which website it is used on.
- The front end of a webpage typically consists of 3 types of interlinked scripts in the following languages:
  - HTML (Hypertext Markup Language): This is the code of the page, it contains its basic structure (paragraphs, links to images and other internal pointers). It's split into the head and the body. The various blocks can be identified by "<block> script <\block>".
  - CSS (Cascading Style Sheets): This language organizes the layout of the page (font, size and colors).
  - JavaScript: These scripts are for the interactive
- The lectures given in this course barely give an introduction, and efficient web scraping would require a better understanding of HTML and CSS. More importantly, the lectures do not give a systematic treatment of the data structures that Python extracts from HTML code.

3.2.1. *Packages and scraping of one webpage.* To practice web scraping, one calls the following packages:

```
import requests
import lxml
import bs4
```

- **Requests:** This package makes requests to access websites and extract their content. Concretely:

  ```
  ini_request = requests.get("https://www.somewebsite.com")
  ```

  the resulting object is a "Response" (a class from the requests package), one attribute of which is "ini_request.text". The source HTML is stored in the latter as one string.
- **Beautiful Soup (bs4):** This package is used to parse the HTML code of the page and to extract the data structures it contains. The first command one uses from this package is:

  ```
  ini_html= bs4.BeautifulSoup(ini_request.text,"lxml")
  ```

  which converts the string "ini_request.text" into data structures that Beautiful Soup can process. Instead of the argument "lxml", we could use bs4's built-in HTML parser with "html.parser", but the lxml library is better optimized for these tasks. To view the HTML code of a webpage with proper indentation etc., one uses the command:

  ```
  print(ini_html.prettify())
  ```

  and another useful command is to extract all text from the webpage:

  ```
  print(ini_html.get_text())
  ```

In the subsections below, we discuss two sandbox examples from a website specifically designed to practice web scraping: books.toscrape.com and quotes.toscrape.com. Since I was quite confused by Pierian Data's data extraction process, I used some functions from bs4's official documentation page:
https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

3.2.2. *Notes.* I do not know when I'll write detailed notes for this part. I however made two Jupyter notebooks with extensive explanations of the scraping process.

- The first example uses the "find_all()" function from bs4, and deals with books on books.toscrape.com (Lectures 121-122).
- The second example uses the "select()" function from bs4, and deals with quotes on books.toscrape.com, which is the assignment of Section 15 (Lectures 123 and 124).