1 - Intro to Python

August 6, 2020

1 Programming for Data Science and Artificial Intelligence

1.1 1 Introduction to Python

1.1.1 Readings: https://docs.python.org/3/tutorial/

- Python code is usually stored in text files with the file ending "`.py`":

  `myprogram.py`

- Comments are made using `#` for single line or `'''` for multiline

- To run our Python program from the command line we use:

  ```
  $ python myprogram.py
  ```

- On UNIX systems it is common to define the path to the interpreter on the first line of the program (note that this is a comment line as far as the Python interpreter is concerned):

  ```
  #!/usr/bin/env python
  ```

  If we do, and if we additionally set the file script to be executable, we can run the program like this:

  ```
  $ myprogram.py
  ```

- This file - an IPython notebook, ending with .ipynb - does not follow the standard pattern with Python code in a text file. Instead, an IPython notebook is stored as a file in the JSON format. The advantage is that we can mix formatted text, Python code and code output. It requires the IPython notebook server to run it though, and therefore isn't a stand-alone Python program as described above. Other than that, there is no difference between the Python code that goes into a program file or an IPython notebook.

- ipython notebook is good for demonstrating, but I personally prefer .py files as it's slightly faster. But anyhow, since we want to combine teaching with python, notebook seems to be the right platform

- Most of the functionality in Python is provided by *modules*. The Python Standard Library is a large collection of modules that provides *cross-platform* implementations of common facilities such as access to the operating system, file I/O, string management, network communication, and much more. To use a module in a Python program it first has to be imported. A module can be imported using the `import` statement. For example, to import the module `math`, which contains many standard mathematical functions, we can do:

1.1.2 Help and importing

[1]:
```
import math
from math import log

print(dir(math))   #list all possible methods
print("\n\n")
help(math.log)  #show build-in function method  #try help(math)  but is very long!
print("log(10): ", log(10))  #support default parameter, here is base e
print("log(10, 2): ", log(10, 2))  #specified parameter, here is base 2
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign',
'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'log2', 'modf', 'nan', 'perm', 'pi', 'pow', 'prod', 'radians',
'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']



Help on built-in function log in module math:

log(…)
    log(x, [base=math.e])
    Return the logarithm of x to the given base.

    If the base not specified, returns the natural logarithm (base e) of x.

log(10):  2.302585092994046
log(10, 2):  3.3219280948873626
```

1.1.3 Basics

[2]:
```
import math

#function is enclosed with () and is called by .
#variables are dynamic typed
x = math.cos(2 * math.pi)  #math.pi is a variable thus there is no ()
hello = "hello"  #declare string
hello2 = 'hello2' #single quote also fine
string_3 = '3'  #this three is a string

#many ways to print
print("1: X with newline: \n", x)  #comma for variable, \n for newline
print("2: X with .format: {}".format(x))  #.format for inserting variable at particular position
```

```
print(f"3: X using f-string with {x}")  #good for not forgetting to write␣
 ↪comma, .format
print("4: User", "Desktop", "Home", sep="/") #with separator
print("5: Hello", end='')  #disable newline
print("6: World")
print(r"7: \n is used for newline")  #use r-string for raw; useful for regular␣
 ↪expressions
print("8: \"Hi")  #use \ to escape "
print("9: C style printing = %f" % 1.0)  #C-style; %s for string, %f for float,␣
 ↪%d for int
print("10: Casting with c style:  value1= %.2f. value2 = %d" % (3.1415, 1.5))

#strings
print("11: hello with +: " + hello)   #concatentation can only be used with␣
 ↪string
print(f"12: hello with f-string: {hello}")
#print("13: X: " + x)  #this will raise errors
#print("14: X + 3: ", x + string_3)  #this will raise errors
print("15: X + 3 with int(): ", x + int(string_3)) #use int() to convert to␣
 ↪string
print("16: 3 with string using str(): ", type(str(int(string_3))))  #use str to␣
 ↪convert back to string, use type() to check type

#boolean, can use either Boolean or int
a = True
a2 = 10  #anything digit not 0 is count as True
b = False
b2 = 0
print("17: Is a boolean: ", type(a) is bool)
print("18: Is a2 int: ", isinstance(a2, int))
print(type(b))
print(type(b2))

if(a):
    print("19: a is True")  #no use of {}, should have equal indentation
if(a2):
    print("20: a2 is True")
if not(b):
    print("21: b is False")
if not(b2):
    print("22: b2 is False")

print("23: True and False: ", True and False)
print("24: True or False: ", True or False)
print("25: Not False: ", not False)
print("26: Not 0: ", not 0)
```

```python

#float + int = float
#int + int = int
#int/float + string = error
print("27: float + int ", type(float(3) + int(3)))
print("28: int + int", type(int(3) + int(3)))
#print("string + int", type(str(3) + int(3))) #error

#math operations
print("29: 3+3 -->", 3 + 3)
print("30: 3/3 -->", 3 / 3)
print("31: 4 // 3 -->", 4 // 3)  #integer division
print("32: 4 mod 3 -->", 4 % 3)  #modulo
print("33: 4 ^ 2 -->", 4 ** 2) #power
x = 1.0 - 1.0j
print(f"34: Complex num: {type(x)} , real: {x.real}, imag: {x.imag}") #complex

#if we do not want to write math.cos, we can import cos
from math import cos
print("35: Without math. ", cos(2* math.pi))
```

```
1: X with newline:
 1.0
2: X with .format: 1.0
3: X using f-string with 1.0
4: User/Desktop/Home
5: Hello6: World
7: \n is used for newline
8: "Hi
9: C style printing = 1.000000
10: Casting with c style:  value1= 3.14. value2 = 1
11: hello with +: hello
12: hello with f-string: hello
15: X + 3 with int():  4.0
16: 3 with string using str():  <class 'str'>
17: Is a boolean:  True
18: Is a2 int:  True
<class 'bool'>
<class 'int'>
19: a is True
20: a2 is True
21: b is False
22: b2 is False
23: True and False:  False
24: True or False:  True
25: Not False:  True
26: Not 0:  True
```

```
27: float + int  <class 'float'>
28: int + int <class 'int'>
29: 3+3 --> 6
30: 3/3 --> 1.0
31: 4 // 3 --> 1
32: 4 mod 3 --> 1
33: 4 ^ 2 --> 16
34: Complex num: <class 'complex'> , real: 1.0, imag: -1.0
35: Without math.  1.0
```

1.1.4 Variable convention

Variable names in Python can contain alphanumerical characters `a-z`, `A-Z`, `0-9` and some special characters such as `_`. Normal variable names must start with a letter.

By convention, variable names start with a lower-case letter, and Class names start with a capital letter.

[3]:
```
a, b = 1, 2  #small letters for variables
PI = 3.14  #it's good practice to use all-caps for constants
#and = 1  #python keyword such as "and" cannot be used as variable name
```

1.1.5 Comparison

[4]:
```
2 > 1, 2 < 1, 2 > 2, 2 < 2, 2 >= 2, 2 <= 2, [1,2] == [1,2]
```

[4]:
```
(True, False, False, False, True, True, True)
```

[5]:
```
# objects identical?
l1 = l2 = [1,2]

l1 is l2
```

[5]:
```
True
```

1.1.6 List and Strings

[6]:
```
string = "String is fun"  #string
list_num = [1, 2, 3, 4] #list of nums
list_string = ["Chaky", "John", "Peter"] #list of strings
list_combine = ["Chaky", 1, 2, True]  #commonly not used, because difficult to␣
 ↪handle

#string/list operations - try this with list_num or list_string
print("1: Length: ", len(string))
print("2: string[0]: ", string[0])
print("3: string[0:2]: ", string[0:2])  #it will exclude the stop
print("4: string[:3]: ", string[:3]) #start beginning with stop 3 thus 0,1,2
```

```
print("5: string[3:]: ", string[3:])  #start with 3 and end wherever thus, 3,␣
 ↪4, 5...
print("6: string[:]: ", string[:]) #everything
print("7: string[-2:]: ", string[-2:]) #from second last until the end, thus␣
 ↪last+1, and last
print("8: string[-0:]: ", string[-0:]) #basically whole thing
print("9: string[::2]: ", string[::2]) #skip every 2
print("10: string[1::2]: ", string[0:5:2])  #from first, skip every 2

#string specific operation
print("11: Replace ing with ange: ", string.replace("ing", "ange"))

#nested list
l1 = [1,[2,3], [4,5], [[6, 7], 8]]
print("12: L1: ", l1)
print("13: Length of L1: ", len(l1))
print("14: l1[3][1]: ", l1[3][1])

#generating list
#range is a iterator object, casting with list return a list
print("15: Range", list(range(10)))  #0 to 9  range(start, stop, step)
print("16: With step 2", list(range(1, 10, 2)))
print("17: With start -10: ", list(range(-5, 5)))

#list is multiplicable
print("18: 5 zeros: ", [0] * 5)

#append and extend list
a = [1, 2]
b = [3]
c = 3
a.append(b)  #append whole list
print("19: A append b", a)
a.extend(b) #extend the original list
print("20: A extend b", a)
a.append(c)
print("21: A append c", a)
#a.extend(c)# error because c is not iterable

#list are mutable
a = [1, 2, 3]
a[0] = 0
print("22: Changing value at index 0 to 0: ", a)
a[0:2] = 5, 6
print("23: Changing using slicing method: ", a)

#getting index
```

```
print("24: Index of 1: ", a.index(5))

#copy()
some_copy = a.copy()
print("25: Comparing copy and original: ", some_copy is a) #False - different memory

#use insert()
l = []
l.insert(0, "i")
l.insert(1, "n")
l.insert(2, "s")
l.insert(3, "e")
l.insert(4, "r")
l.insert(5, "t")
print("26: Insert: ", "".join(l))  #"" specifies the separator

#use remove() and pop()
l.remove("i")
print("27: l removing i", l)
l.pop(0)  #remove value from index
print("28: l removing index 0", l)

#sort
a = [3,19,5]
b = ["Car", "Apple", "Boy"]
a.sort()
b.sort()  #try put argument reverse=True
print("29: Sorted [3, 19, 5]: ", a)
print("30: Sorted ['Car', 'Apple', 'Boy']: ", b)
print("31: Reverse: ", b.reverse(), b)  #remember that b.reverse or sort is in place, and return None
a.insert(1, 4)
print("32: A.insert(1,4)", a)  #this will shift to the right

#check things in list
print("33: 3 in a?", 3 in a)
print("34: 99 in a?", 4 in a)

#looping list
for i in range(10): #does not include 10, start with 0
    print("35: I: ", i)

#given a list
some_list = [8, 9, 10]

for num in some_list:  #use in
```

```
    print("36:", num)

for i in range(len(some_list)):  #use len (old guys love to do this!)
    print("37:", some_list[i])

for ix, (value) in enumerate(some_list):
    print(f"38: Index: {ix}; value: {value}")

# Making random_list is also a common task in research
import random

some_rand_list = []

for i in range(10):
    some_rand_list.append(random.randint(1, 10))  #randint(a, b) return random␣
 ↪integer N where a<=N<=b
print("39: Some random list: ", some_rand_list)

#Another way to random is to use shuffle
a = list(range(10))
random.shuffle(a)  #this is in place!  so it will return None!
print("40: Suffled list: ", a)
print("41: Sampled 3 from a list: ", random.sample(a, 3))
print("42: Choose 1 randomly from a list: ", random.choice(a))

#choice is useful for words, e.g.,
my_dictionary = ['fridge', 'pen', 'apple']
print("43: Random choice to put in your algorithm: ", random.
 ↪choice(my_dictionary))

#permuations using itertools
from itertools import permutations, combinations
print("44: All possible permuations: ", list(permutations(['1','2','3'])))
print("45: All possible pairs, ordered-sensitive: ",␣
 ↪list(permutations(['1','2','3'],2)))
print("46: All possible pairs, not ordered-sensitive: ",␣
 ↪list(combinations(['1','2','3'], 2)))


#you will love this - list comprehension
# [ expression for item in list if conditional ]
#let's say i have this list
x = [1, 2, 3]
#I want to +1 to every element to the list, I can do
for i in range(len(x)):
    x[i] +=1
print("47:", x)
```

```
#better yet - use list comprehension
new_x = [num + 1 for num in x]  #remember to enclose with []!!
print("48:", new_x)

#let's pokemon
#suppose we have two lists - one are pokemon names, and another are their hps
names = ['Bulbasaur', 'Charmander', 'Squirtle']
hps = [45, 49, 44]

#we can use non-pythonic way (i..e, not recommended) to combine them as tuples
combined = []
for i, pokemon in enumerate(names):
    combined.append((pokemon, hps[i]))

print("49: Pokemon stats - non-pythonic: ", combined)

#more pythonic is to use zip()
#zip: The zip method is used to combine multiple lists in Python
#into tuples. If the two lists are not the same length,
#then the longer of the two lists would be truncated to the length
#of the shorter.
print("50: Pokemon stats - pythonic: ", list(zip(names, hps)))  #zip is␣
 ↪iterable, not list or dict

#we can use * to unpack zip to tuples and use [] to convert to list
print("51: Pokemon stats - pythonic2: ", [*zip(names, hps)]    )  #zip is␣
 ↪iterable, not list or dict

#let's learn about set
list_a = ['Bulbasaur', 'Charmander', 'Squirtle', 'Charmander']
list_b = ['Caterpie', 'Pidgey', 'Squirtle']
set_a = set(list_a)
print("52: Set a - i.e., no duplicates: ", set_a)
set_b = set(list_b)

#intersect; make sure to convert list to set first!
print("53: Set intersection: ", set_a.intersection(set_b))
print("54: Set a - b: ", set_a.difference(set_b))
print("55: Set b - a: ", set_b.difference(set_a))
print("56: Set (a-b) + (b-a): ", set_a.symmetric_difference(set_b))
print("57: Set (a-b) + (b-a): ", set_b.symmetric_difference(set_a)) #same␣
 ↪result both sides
print("58: Set a U b", set_a.union(set_b))  #remove any duplicate intersection


#let's try more real-world, advanced list
two_channel_eeg_signal1 = [8, 9]
```

```
event1 = 1
two_channel_eeg_signal2 = [3, 3]
event2 = 2
two_channel_eeg_signal3 = [2, 3]
event3 = 2
some_nested_list = []
some_nested_list.append([two_channel_eeg_signal1, event1])
some_nested_list.append([two_channel_eeg_signal2, event2])
some_nested_list.append([two_channel_eeg_signal3, event3])
print("59: EEG signal: ", some_nested_list)

#how do i get only the eeg_signal, and only the events?
eeg = []
event = []
for each_event_list in some_nested_list:
    each_eeg = each_event_list[0]
    each_event = each_event_list[1]
    eeg.append(each_eeg)
    eeg.append(each_event)
    print("60: EEG: ", each_eeg)
    print("61: Event: ", each_event)

#use list comprenhension
print("62: Use list comprehension EEG: ", [i[0] for i in some_nested_list])
print("63: Use list comprehension Event: ", [i[1] for i in some_nested_list])

#how to get only event 2 eeg signal?
print("64: Use list comprehension EEG: ", [i[0] for i in some_nested_list if␣
 ↪i[1]==2])
```

```
1: Length:  13
2: string[0]:  S
3: string[0:2]:  St
4: string[:3]:  Str
5: string[3:]:  ing is fun
6: string[:]:  String is fun
7: string[-2:]:  un
8: string[-0:]:  String is fun
9: string[::2]:  Srn sfn
10: string[1::2]:  Srn
11: Replace ing with ange:  Strange is fun
12: L1:  [1, [2, 3], [4, 5], [[6, 7], 8]]
13: Length of L1:  4
14: l1[3][1]:  8
15: Range [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
16: With step 2 [1, 3, 5, 7, 9]
17: With start -10:  [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
```

```
18: 5 zeros:  [0, 0, 0, 0, 0]
19: A append b [1, 2, [3]]
20: A extend b [1, 2, [3], 3]
21: A append c [1, 2, [3], 3, 3]
22: Changing value at index 0 to 0:  [0, 2, 3]
23: Changing using slicing method:  [5, 6, 3]
24: Index of 1:  0
25: Comparing copy and original:  False
26: Insert:  insert
27: l removing i ['n', 's', 'e', 'r', 't']
28: l removing index 0 ['s', 'e', 'r', 't']
29: Sorted [3, 19, 5]:  [3, 5, 19]
30: Sorted ['Car', 'Apple', 'Boy']:  ['Apple', 'Boy', 'Car']
31: Reverse:  None ['Car', 'Boy', 'Apple']
32: A.insert(1,4) [3, 4, 5, 19]
33: 3 in a? True
34: 99 in a? True
35: I:  0
35: I:  1
35: I:  2
35: I:  3
35: I:  4
35: I:  5
35: I:  6
35: I:  7
35: I:  8
35: I:  9
36: 8
36: 9
36: 10
37: 8
37: 9
37: 10
38: Index: 0; value: 8
38: Index: 1; value: 9
38: Index: 2; value: 10
39: Some random list:  [8, 6, 7, 10, 9, 10, 2, 1, 8, 2]
40: Suffled list:  [6, 7, 3, 2, 0, 9, 5, 1, 4, 8]
41: Sampled 3 from a list:  [4, 0, 7]
42: Choose 1 randomly from a list:  8
43: Random choice to put in your algorithm:  apple
44: All possible permuations:  [('1', '2', '3'), ('1', '3', '2'), ('2', '1',
'3'), ('2', '3', '1'), ('3', '1', '2'), ('3', '2', '1')]
45: All possible pairs, ordered-sensitive:  [('1', '2'), ('1', '3'), ('2', '1'),
('2', '3'), ('3', '1'), ('3', '2')]
46: All possible pairs, not ordered-sensitive:  [('1', '2'), ('1', '3'), ('2',
'3')]
47: [2, 3, 4]
```

```
48: [3, 4, 5]
49: Pokemon stats - non-pythonic:  [('Bulbasaur', 45), ('Charmander', 49),
('Squirtle', 44)]
50: Pokemon stats - pythonic:  [('Bulbasaur', 45), ('Charmander', 49),
('Squirtle', 44)]
51: Pokemon stats - pythonic2:  [('Bulbasaur', 45), ('Charmander', 49),
('Squirtle', 44)]
52: Set a - i.e., no duplicates:  {'Charmander', 'Squirtle', 'Bulbasaur'}
53: Set intersection:  {'Squirtle'}
54: Set a - b:  {'Charmander', 'Bulbasaur'}
55: Set b - a:  {'Caterpie', 'Pidgey'}
56: Set (a-b) + (b-a):  {'Charmander', 'Bulbasaur', 'Caterpie', 'Pidgey'}
57: Set (a-b) + (b-a):  {'Charmander', 'Caterpie', 'Pidgey', 'Bulbasaur'}
58: Set a U b {'Charmander', 'Pidgey', 'Squirtle', 'Caterpie', 'Bulbasaur'}
59: EEG signal:  [[[8, 9], 1], [[3, 3], 2], [[2, 3], 2]]
60: EEG:  [8, 9]
61: Event:  1
60: EEG:  [3, 3]
61: Event:  2
60: EEG:  [2, 3]
61: Event:  2
62: Use list comprehension EEG:  [[8, 9], [3, 3], [2, 3]]
63: Use list comprehension Event:  [1, 2, 2]
64: Use list comprehension EEG:  [[3, 3], [2, 3]]
```

1.1.7 Tuples

Tuples are like lists, except that they cannot be modified once created, that is they are *immutable*.

1. You can't add to tuples, so no extend or append
2. You can't remove or insert, so no insert, remove, pop
3. You can find, thus "in" or indexing can be used
4. Tuples are much faster than list

Make sense to use tuples for write-protected data

In Python, tuples are created using the syntax `(..., ..., ...)`, or even `..., ...`:

[7]:
```
#creating tuples
point = (10, 20)
point2 = 10, 20
print(point, type(point))
print(point2, type(point2))
```

```
(10, 20) <class 'tuple'>
(10, 20) <class 'tuple'>
```

[8]:
```
#unpack tuples to individual variable
x, y = point
print("x =", x)
```

```
print("y =", y)
```

```
x = 10
y = 20
```

[9]:
```
#point[0] = 20 #errors
```

1.1.8 Dictionaries

Dictionaries are also like lists, except that each element is a key-value pair. The syntax for dictionaries is `{key1 : value1, ...}`:

[10]:
```
some_dict = {"Pineapple" : 12,
             "Orange" : 10,
             "Apple" : 15,}

print(type(some_dict))
print(some_dict)
```

```
<class 'dict'>
{'Pineapple': 12, 'Orange': 10, 'Apple': 15}
```

[11]:
```
#accessing the value with key
print("Pineapple = " + str(some_dict["Pineapple"]))
print("Orange = " + str(some_dict["Orange"]))
print("Apple = " + str(some_dict["Apple"]))

#adding new key and value
some_dict["Durian"] = 50
```

```
Pineapple = 12
Orange = 10
Apple = 15
```

[12]:
```
#Looping in dict...
for key in some_dict:
    print("1:", key + " = " + str(some_dict[key]))

#With items()
for key, value in some_dict.items():
    print("2:", key, value)

#With sorted()
for key, value in sorted(some_dict.items()):
    print("3:", key, value)

#With enumerate...remember that it will return as tuples for the pair
for index, (key, value) in enumerate(some_dict.items()):
```

```
    print("4:", index, key, value)


#zip is useful to perform a parallel iteration over multiple dict
dict_one = {'name': 'John', 'last_name': 'Doe', 'job': 'Python Consultant'}
dict_two = {'name': 'Jane', 'last_name': 'Doe', 'job': 'Community Manager'}
for (k1, v1), (k2, v2) in zip(dict_one.items(), dict_two.items()):
    print("5:", k1, '->', v1)
    print("6:", k2, '->', v2)
```

```
1: Pineapple = 12
1: Orange = 10
1: Apple = 15
1: Durian = 50
2: Pineapple 12
2: Orange 10
2: Apple 15
2: Durian 50
3: Apple 15
3: Durian 50
3: Orange 10
3: Pineapple 12
4: 0 Pineapple 12
4: 1 Orange 10
4: 2 Apple 15
4: 3 Durian 50
5: name -> John
6: name -> Jane
5: last_name -> Doe
6: last_name -> Doe
5: job -> Python Consultant
6: job -> Community Manager
```

1.1.9 If, while

[13]:
```
statement1 = False
statement2 = False
statement3 = True

#use indentation, this is one annoying aspect of python!
if statement1:
    if statement3:
        print("1: statement1 is True")

elif statement2:
    print("2: statement2 is True")
else:
```

```
    print("3: statement1 and statement2 are False")

    print("4: Still inside the else block!!")

print("5: Now outside the if block!")


i = 0
while i < 5:
    print("6:", i)
    i = i + 1  #try i+=1
print("7: done")
```

```
3: statement1 and statement2 are False
4: Still inside the else block!!
5: Now outside the if block!
6: 0
6: 1
6: 2
6: 3
6: 4
7: done
```

1.1.10 Functions

A function in Python is defined using the keyword `def`, followed by a function name, a signature within parentheses `()`, and a colon `:`. The following code, with one additional level of indentation, is the function body.

[14]:
```
def func0():
    print("test")
```

[15]:
```
func0()
```

```
test
```

Optionally, but highly recommended, we can define a so called "docstring", which is a description of the functions purpose and behaivor. The docstring should follow directly after the function definition, before the code in the function body.

[16]:
```
def func1(s):
    """
    Print a string 's' and tell how many characters it has
    """

    print(s + " has " + str(len(s)) + " characters")
```

[17]:
```
help(func1)
```

```
Help on function func1 in module __main__:

func1(s)
    Print a string 's' and tell how many characters it has
```

[18]:
```
func1("test")
```

```
test has 4 characters
```

Functions that returns a value use the `return` keyword:

[19]:
```
def square(x):
    """
    Return the square of x.
    """
    return x ** 2
```

[20]:
```
square(4)
```

[20]: 16

We can return multiple values from a function using tuples (see above):

[21]:
```
def powers(x):
    """
    Return a few powers of x.
    """
    return x ** 2, x ** 3, x ** 4
```

[22]:
```
powers(3)
```

[22]: (9, 27, 81)

[23]:
```
x2, x3, x4 = powers(3)

print(x3)
```

```
27
```

1.1.11 Default argument and keyword arguments

In a definition of a function, we can give default values to the arguments the function takes:

[24]:
```
def myfunc(x, p=2, debug=False):
    if debug:
        print("evaluating myfunc for x = " + str(x) + " using exponent p = " +␣
↪str(p))
    return x**p
```

If we don't provide a value of the `debug` argument when calling the the function `myfunc` it defaults to the value provided in the function definition:

[25]:
```
myfunc(5)
```

[25]:
```
25
```

[26]:
```
myfunc(5, debug=True)
```

```
evaluating myfunc for x = 5 using exponent p = 2
```

[26]:
```
25
```

If we explicitly list the name of the arguments in the function calls, they do not need to come in the same order as in the function definition. This is called *keyword* arguments, and is often very useful in functions that takes a lot of optional arguments.

[27]:
```
myfunc(p=3, debug=True, x=7)  #argument name must match that of the func!
```

```
evaluating myfunc for x = 7 using exponent p = 3
```

[27]:
```
343
```

1.1.12 Unnamed functions (lambda function)

In Python we can also create unnamed functions, using the `lambda` keyword:

[28]:
```
f1 = lambda x: x**2

# is equivalent to

def f2(x):
    return x**2
```

[29]:
```
f1(2), f2(2)
```

[29]:
```
(4, 4)
```

This technique is useful for example when we want to pass a simple function as an argument to another function, like this:

[30]:
```
#Python map() function is used to apply a function on all the elements
#of specified iterable and return map object. Python map object is an iterator,
#so we can iterate over its elements
list(map(lambda x: x**2, range(-3,4)))   #map(function, iterables)
```

[30]:
```
[9, 4, 1, 0, 1, 4, 9]
```

1.1.13 Classes

Classes are the key features of object-oriented programming. A class is a structure for representing an object and the operations that can be performed on the object.

In Python a class can contain *attributes* (variables) and *methods* (functions).

A class is defined almost like a function, but using the `class` keyword, and the class definition usually contains a number of class method definitions (a function in a class).

- **Each class method should have an argument `self` as it first argument. This object is a self-reference.**
- Some class method names have special meaning, for example:
  - `__init__`: The name of the method that is invoked when the object is first created.
  - `__str__` : A method that is invoked when a simple string representation of the class is needed, as for example when printed.
  - There are many more, see http://docs.python.org/2/reference/datamodel.html#special-method-names

[31]:
```
class Point:
    """
    Simple class for representing a point in a Cartesian coordinate system.
    """

    def __init__(self, x, y):
        """
        Create a new Point at x, y.
        """
        self.x = x
        self.y = y

    def translate(self, dx, dy):
        """
        Translate the point by dx and dy in the x and y direction.
        """
        self.x += dx
        self.y += dy

    def __str__(self):
        return("Point at [%f, %f]" % (self.x, self.y))
```

To create a new instance of a class:

[32]:
```
p1 = Point(0, 0) # this will invoke the __init__ method in the Point class

print(p1)         # this will invoke the __str__ method
```

```
Point at [0.000000, 0.000000]
```

To invoke a class method in the class instance `p`:

[33]:
```
p2 = Point(1, 1)

p1.translate(0.25, 1.5)

print(p1)
print(p2)
```

```
Point at [0.250000, 1.500000]
Point at [1.000000, 1.000000]
```

1.2 Modules

One of the most important concepts in good programming is to reuse code and avoid repetitions.

The idea is to write functions and classes with a well-defined purpose and scope, and reuse these instead of repeating similar code in different part of a program (modular programming). The result is usually that readability and maintainability of a program is greatly improved. What this means in practice is that our programs have fewer bugs, are easier to extend and debug/troubleshoot.

Python supports modular programming at different levels. Functions and classes are examples of tools for low-level modular programming. Python modules are a higher-level modular programming construct, where we can collect related variables, functions and classes in a module. A python module is defined in a python file (with file-ending `.py`), and it can be made accessible to other Python modules and programs using the `import` statement.

Consider the following example: the file `mymodule.py` contains simple example implementations of a variable, function and a class:

[34]:
```
%%file resources/mymodule.py
#%%file is used to write whatever content here into mymodule.py
"""
Example of a python module. Contains a variable called my_variable,
a function called my_function, and a class called MyClass.
"""

my_variable = 0

def my_function():
    """
    Example function
    """
    return my_variable

class MyClass:
    """
    Example class.
    """

    def __init__(self):
```

```
        self.variable = my_variable

    def set_variable(self, new_value):
        """
        Set self.variable to a new value
        """
        self.variable = new_value

    def get_variable(self):
        return self.variable

#common question is what is __name_ == "__main__"?
if __name__ == "__main__":
    print("Executed when invoked directly")
else:
    print("Executed when imported")
```

```
Overwriting resources/mymodule.py
```

We can import the module `mymodule` into our Python program using `import`:

[35]:
```
import sys
sys.path.insert(1, 'resources/')

import mymodule
```

```
Executed when imported
```

Use `help(module)` to get a summary of what the module provides:

[36]:
```
help(mymodule)
```

```
Help on module mymodule:

NAME
    mymodule

DESCRIPTION
    Example of a python module. Contains a variable called my_variable,
    a function called my_function, and a class called MyClass.

CLASSES
    builtins.object
        MyClass

    class MyClass(builtins.object)
     |  Example class.
     |
     |  Methods defined here:
```

```
 |
 |  __init__(self)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  get_variable(self)
 |
 |  set_variable(self, new_value)
 |      Set self.variable to a new value
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

FUNCTIONS
    my_function()
        Example function

DATA
    my_variable = 0

FILE
    /home/akrarads/_AIT/2020_pythonDSAI/Python-for-DS-AI/resources/mymodule.py
```

[37]:
```
mymodule.my_variable
```

[37]:
```
0
```

[38]:
```
mymodule.my_function()
```

[38]:
```
0
```

[39]:
```
my_class = mymodule.MyClass()
my_class.set_variable(10)
my_class.get_variable()
```

[39]:
```
10
```

If we make changes to the code in `mymodule.py`, we need to reload it using `reload`:

[40]:
```
#in python3 reload has been moved to importlib
from importlib import reload
reload(mymodule)
```

```
Executed when imported
```

[40]:
```
<module 'mymodule' from 'resources/mymodule.py'>
```

1.3 Exceptions

In Python errors are managed with a special language construct called "Exceptions". When errors occur exceptions can be raised, which interrupts the normal program flow and fallback to somewhere else in the code where the closest try-except statement is defined.

To generate an exception we can use the `raise` statement, which takes an argument that must be an instance of the class `BaseException` or a class derived from it.

[41]:
```
raise Exception("description of the error")
```

```
␣
↪---------------------------------------------------------------------------

Exception                                 Traceback (most recent call␣
↪last)

<ipython-input-41-c32f93e4dfa0> in <module>
----> 1 raise Exception("description of the error")


Exception: description of the error
```

A typical use of exceptions is to abort functions when some error condition occurs, for example:

```
def my_function(arguments):

    if not verify(arguments):
        raise Exception("Invalid arguments")

    # rest of the code goes here
```

To gracefully catch errors that are generated by functions and class methods, or by the Python interpreter itself, use the `try` and `except` statements:

```
try:
    # normal code goes here
except:
    # code for error handling goes here
    # this code is not executed unless the code
```

```
    # above generated an error
```

For example:

[42]:
```
try:
    print("test")
    # generate an error: the variable test is not defined
    print(test)
except:
    print("Caught an exception")
```

```
test
Caught an exception
```

To get information about the error, we can access the `Exception` class instance that describes the exception by using for example:

`except Exception as e:`

[43]:
```
try:
    print("test")
    # generate an error: the variable test is not defined
    print(test)
except Exception as e:
    print("Caught an exception: " + str(e))
```

```
test
Caught an exception: name 'test' is not defined
```

1.3.1 Reading, creating file

[44]:
```
#r open file for reading ONLY
#r+ open file for boht reading and writing
#w open file for writing ONLY, overwrite everything if exists, or create the␣
 ↪file if not exists
#w+ open a file for both writing and reading.  Overwrites if exist, otherwise,␣
 ↪creates a new file for reading and writing
#a - pointer at end of file.  Create file if does not exists
#a+ - two pointers - one at beginning for read, and another at end for write

import os

directory = "resources"  #use forward slashes in pathlib
filename = os.path.join(directory, "test.txt")

#file = open("test.txt", "r") #error file not exists

f = open(filename, "w+")
f.write("Hello World\nSecond Line")  #writing contents
```

```
f.seek(0)  #since after writing, f pointer is at end of file
contents = f.read()
f.seek(0) #same here, reset the pointer
line_content = f.readlines()
print("Reading contents: ", contents)
print("Reading line_contents: ", line_content)
f.close() #save memory

f = open(filename, "a+")
f.write("\nAppended line")
f.seek(0)  #since after writing, f pointer is at end of file
contents = f.read()
print("Reading contents: ", contents)
f.close()

f = open(filename, "r+")
print("Reading only: ", f.read())

#finding files based on some expression
#useful for image/signal algorithms
#glob is a technique of retrieving files/pathnames matching a specified pattern
#with glob, we can use *, ? ([ranges])
import glob

# Using '?' pattern
print('\nNamed with ?:')
for name in glob.glob('resources/letter_highlight_images/?_black.png'):
    print(name)

print('\nNamed with range:')
for name in glob.glob('resources/letter_highlight_images/[A-C]_black.png'):
    print(name)

# Using '?' pattern
print('\nUsing recursive with ** and recursive = True:')
for name in glob.glob('resources/letter_highlight_images/**/?_black.png',␣
 ↪recursive=True):
    print(name)
```

```
Reading contents:  Hello World
Second Line
Reading line_contents:  ['Hello World\n', 'Second Line']
Reading contents:  Hello World
Second Line
Appended line
Reading only:  Hello World
Second Line
```

```
Appended line

Named with ?:
resources/letter_highlight_images/Q_black.png
resources/letter_highlight_images/M_black.png
resources/letter_highlight_images/U_black.png
resources/letter_highlight_images/Z_black.png
resources/letter_highlight_images/Y_black.png
resources/letter_highlight_images/C_black.png
resources/letter_highlight_images/D_black.png
resources/letter_highlight_images/P_black.png
resources/letter_highlight_images/B_black.png
resources/letter_highlight_images/V_black.png
resources/letter_highlight_images/A_black.png
resources/letter_highlight_images/O_black.png
resources/letter_highlight_images/X_black.png
resources/letter_highlight_images/R_black.png
resources/letter_highlight_images/W_black.png
resources/letter_highlight_images/L_black.png
resources/letter_highlight_images/T_black.png
resources/letter_highlight_images/N_black.png
resources/letter_highlight_images/E_black.png
resources/letter_highlight_images/S_black.png
resources/letter_highlight_images/F_black.png
resources/letter_highlight_images/G_black.png

Named with range:
resources/letter_highlight_images/C_black.png
resources/letter_highlight_images/B_black.png
resources/letter_highlight_images/A_black.png

Using recursive with ** and recursive = True:
resources/letter_highlight_images/Q_black.png
resources/letter_highlight_images/M_black.png
resources/letter_highlight_images/U_black.png
resources/letter_highlight_images/Z_black.png
resources/letter_highlight_images/Y_black.png
resources/letter_highlight_images/C_black.png
resources/letter_highlight_images/D_black.png
resources/letter_highlight_images/P_black.png
resources/letter_highlight_images/B_black.png
resources/letter_highlight_images/V_black.png
resources/letter_highlight_images/A_black.png
resources/letter_highlight_images/O_black.png
resources/letter_highlight_images/X_black.png
resources/letter_highlight_images/R_black.png
resources/letter_highlight_images/W_black.png
resources/letter_highlight_images/L_black.png
```

```
resources/letter_highlight_images/T_black.png
resources/letter_highlight_images/N_black.png
resources/letter_highlight_images/E_black.png
resources/letter_highlight_images/S_black.png
resources/letter_highlight_images/F_black.png
resources/letter_highlight_images/G_black.png
resources/letter_highlight_images/some_folder/K_black.png
resources/letter_highlight_images/some_folder/H_black.png
resources/letter_highlight_images/some_folder/I_black.png
resources/letter_highlight_images/some_folder/J_black.png
```

1.3.2 Logging

[45]:
```
import logging

logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')

logfile = os.path.join("resources", "development.log")

#Create a custom logger
logger = logging.getLogger(__name__)  #__name__ corresponds to where execution␣
↪starts

# Create handlers
c_handler = logging.StreamHandler()  #prints to console
f_handler = logging.FileHandler(logfile) #prints to file
c_handler.setLevel(logging.WARNING) #all warning will be handled by c
f_handler.setLevel(logging.ERROR) #all errors handled by f

# Create formatters and add it to handlers
c_format = logging.Formatter('%(name)s - %(levelname)s - %(message)s')
f_format = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s -␣
↪%(message)s')
c_handler.setFormatter(c_format)
f_handler.setFormatter(f_format)

# Add handlers to the logger
logger.addHandler(c_handler)
logger.addHandler(f_handler)

logger.warning('This is a warning')
logger.error('This is an error')
```

```
#logging variable error
name = "Chaky"
logger.error("%s raised an error", name)

#logging exception
a = 1
b = 0
try:
    c = a / b
except Exception as e:
    logger.error("Exception occured", exc_info=True)

#all attributes can be found here - https://docs.python.org/3/library/logging.
 ↪html#logrecord-attributes
```

```
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical message
__main__ - WARNING - This is a warning
WARNING:__main__:This is a warning
__main__ - ERROR - This is an error
ERROR:__main__:This is an error
__main__ - ERROR - Chaky raised an error
ERROR:__main__:Chaky raised an error
__main__ - ERROR - Exception occured
Traceback (most recent call last):
  File "<ipython-input-45-61a38891412c>", line 41, in <module>
    c = a / b
ZeroDivisionError: division by zero
ERROR:__main__:Exception occured
Traceback (most recent call last):
  File "<ipython-input-45-61a38891412c>", line 41, in <module>
    c = a / b
ZeroDivisionError: division by zero
```

1.3.3 Progress bar while looping

[46]:
```
from tqdm import tqdm

for i in tqdm(range(10000)):
    pass
```

```
100%|      | 10000/10000 [00:00<00:00, 4115290.42it/s]
```

1.4 Multiprocessing vs. threading

Python is designed to run sequentially. Multiprocessing and threading are two common ways to run asynchrously in the background.

Threading use threads, while multiprocessing uses processes. Threading is much more light weight since threads share same memory space, while processes do not.

When to use threading: - Network or I/O related task. Typical scenario is when you are monitoring multiple sources of data in parallel. Let's say you are reading human eeg signal, at the same time, reading human heart rate signal, threading is game-changing. It allows you to asynchronously read data from various sources and combine them later on.

Big cons of threading: - difficult to debug, can never ensure who goes first - possible concurrency problem since threads are shared. Require implementation of locks

When to use multiprocessing: - CPU intensive task. Python is not designed to utilize the computer max spec. Multiprocessing here comes to the rescue. No problem with concurrency since memory are not shared.

Big cons of multiprocessing: - Big overhead shuffling around processes

1.4.1 Multiprocessing

[47]:
```
# importing the multiprocessing module
import multiprocessing
import time
import os

def first_func(num):
    print("First func start time: ", time.time())
    print("Cube: {}".format(num ** 3))
    print("PID: ", os.getpid())

def second_func(num):
    print("Second func start time: ", time.time())
    print("Square: {}".format(num ** 2))
    print("PID: ", os.getpid())


#creating processes
p1 = multiprocessing.Process(target=first_func, args=(10, ))
p2 = multiprocessing.Process(target=second_func, args=(10, ))

# starting process 1
p1.start()
# starting process 2
p2.start()

# wait until process 1 is finished
p1.join()
# wait until process 2 is finished
p2.join()
```

```
# both processes finished
print("Done!")

#without multiprocesses
first_func(10)
second_func(10)
```

```
First func start time:1591168159.1299107Second func start time:
Cube: 10001591168159.1351159
PID:95527

Square: 100
PID:95530
Done!
First func start time:  1591168159.1554983
Cube: 1000
PID:  95495
Second func start time:  1591168159.1556811
Square: 100
PID:  95495
```

1.4.2 Threading

[48]:
```
import threading

def first_func(num):
    print("First func start time: ", time.time())
    print("Cube: {}".format(num ** 3))
    print("PID: ", os.getpid())

def second_func(num):
    print("Second func start time: ", time.time())
    print("Square: {}".format(num ** 2))
    print("PID: ", os.getpid())

#creating threads
p1 = threading.Thread(target=first_func, args=(10, ))
p2 = threading.Thread(target=second_func, args=(10, ))

# starting process 1
p1.start()
# starting process 2
p2.start()

# wait until process 1 is finished
p1.join()
# wait until process 2 is finished
```

```
p2.join()

# both processes finished
print("Done!")

#without multiprocesses
first_func(10)
second_func(10)
```

```
First func start time:  1591168159.1860304
Cube: 1000
PID:  95495
Second func start time:  1591168159.1868515
Square: 100
PID:  95495
Done!
First func start time:  1591168159.1892362
Cube: 1000
PID:  95495
Second func start time:  1591168159.1899846
Square: 100
PID:  95495
```

1.4.3 Timing code run time

[49]:
```
import timeit #also check out line_profiler and memory_profiler
timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)  #iterates␣
↪10000 times
```

[49]:
```
0.11789018499257509
```

[ ]:
```

```