day_1_lecture

January 8, 2020

1 Day 1: Python Basic

1.1 Hello World!

In Python we can display a text with the `print()` function.

```
In [1]: print('Hello World')
```

```
Hello World
```

In Jupyter notebooks, we can also leave the `print()` function away. The jupyter notebook recognizes single statements which otherwise would be useless and prints them. Also if values are returned, they are printed.

```
In [2]: 'Hello World'
```

```
Out[2]: 'Hello World'
```

1.2 Variables

In Python we can simply asign values or objects to a variable with '='

```
In [3]: x = 5
        print(x)
```

```
5
```

In Python, the type of the variable is not permanent and can change during run time (Different than from Java, C++ and others). The built in function *type()* reveals the type of a variable. It is **very** useful!!!

```
In [4]: x = 5
        print(type(x))

        x = 'five'
        print(type(x))
```

```
<class 'int'>
<class 'str'>
```

Other common used simple types for variables are:

```
In [5]: a = 1.1
        b = True
        c = [1,2,3]
        d = (1,2,3)
        type(a), type(b), type(c), type(d)

Out[5]: (float, bool, list, tuple)
```

2 Operators, Types and Casting

Addition, subtraction and multiplication of integers dont change the resulting type.

```
In [6]: type(4*5), type(4+5), type(4-10)

Out[6]: (int, int, int)
```

Diviosion does change the type!

```
In [7]: 4/3, type(4.0), type(3.0), type(4/3)

Out[7]: (1.3333333333333333, float, float, float)
```

Also even if the division actually is an integer!

```
In [8]: 4/2, type(4), type(2), type(4/2)

Out[8]: (2.0, int, int, float)
```

With casting, we can change the type of one variable in to another.

```
In [9]: int(4.0/3.0), 4//3, type(4//3), int(4.0)/int(3.0) #changing floats to ints, gotta respec

Out[9]: (1, 1, int, 1.3333333333333333)
```

We can also cast booleans:

```
In [10]: int(True), int(False),bool(0),bool(1), bool(100000000), bool(0.00000001), bool(-1)

Out[10]: (1, 0, False, True, True, True, True)
```

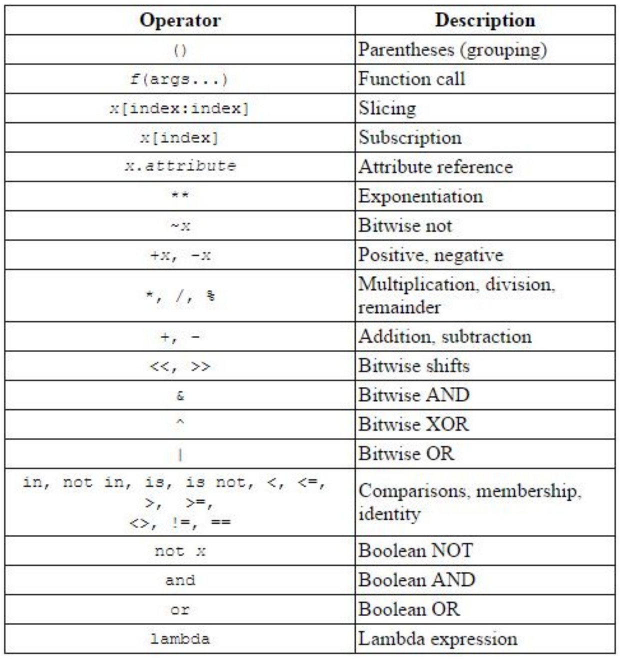
But not every form of casting is always possible:

```
In [11]: int("50")

Out[11]: 50
```

```
In [12]: int("fifty")


        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-12-a85df3e87dfe> in <module>
    ----> 1 int("fifty")


        ValueError: invalid literal for int() with base 10: 'fifty'


In [13]: float("50.5")

Out[13]: 50.5

In [14]: str(4), str(5), bool("Hello"), bool("False"), bool(""),

Out[14]: ('4', '5', True, True, False)
```

Operators can be applied to more complex types of objects, and the way they apply depend on these types:

```
In [15]: 1 + 2

Out[15]: 3

In [16]: [1, 2, 3] + [3, 2, 1]

Out[16]: [1, 2, 3, 3, 2, 1]

In [17]: (1,2)+(2,3)

Out[17]: (1, 2, 2, 3)

In [18]: [1, 2, 4] * [1, 3]


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-18-a31fbedbf558> in <module>
    ----> 1 [1, 2, 4] * [1, 3]


        TypeError: can't multiply sequence by non-int of type 'list'
```



Operator	Description
`()`	Parentheses (grouping)
`f(args...)`	Function call
`x[index:index]`	Slicing
`x[index]`	Subscription
`x.attribute`	Attribute reference
`**`	Exponentiation
`~x`	Bitwise not
`+x, -x`	Positive, negative
`*, /, %`	Multiplication, division, remainder
`+, -`	Addition, subtraction
`<<, >>`	Bitwise shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`in, not in, is, is not, <, <=, >, >=, <>, !=, ==`	Comparisons, membership, identity
`not x`	Boolean NOT
`and`	Boolean AND
`or`	Boolean OR
`lambda`	Lambda expression

3 Booleans

```
In [19]: a = True
         not a

Out[19]: False

In [ ]:

In [20]: True or False, True and False, not (True or False)

Out[20]: (True, False, False)

In [21]: 2 == 2.0, 2 == 4, 2 != 4, 2 is not 4

Out[21]: (True, False, True, True)

In [ ]:

In [22]: "hello" is "world", "hello" is 'hello'

Out[22]: (False, True)

In [23]: "hello" is 33

Out[23]: False
```

4 If - Else

With if statements, we can run some code if a certain condition is met

```
In [24]: a = True
         if a:
             print("a is True")
         else:
             print("a is False")
```

```
a is True
```

To check multiple conditions, we can also write if statments in if, and use elif . When using if and elif, only one statement will be fullfilled.

```
In [25]: b = 5

         if b == 2:
             print("b is 2")
         elif b == 3:
             print("b is 3")
         elif b == "sarah":
             print("b is sarah")
         else:
             print("b is something else")
```

```
b is something else
```

```
In [26]: if a is not b:
             print("a is not b")
```

```
a is not b
```

5 Strings

We can write text in quotation marks to create a string

```
In [27]: a = "this is a string"
         a

Out[27]: 'this is a string'
```

Concatenation of string can be simpy done with '+'

```
In [28]: "hello" + " " + "world"

Out[28]: 'hello world'
```

Strings can be also be repeated

```
In [29]: 3 * "Python"

Out[29]: 'PythonPythonPython'
```

6 Lists

Lists in python can be created in multiple ways:

```
In [30]: list_1 = [0,1,2,3,4,5,6,7,8,9]
         list_1

Out[30]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [31]: list_0 = list()
         list_0

Out[31]: []
```

List can also be created with repetetion

```
In [32]: list_2 = [1]*5
         print(list_2)
         list_3 = [1, 4, 8] *3
         print(list_3)

[1, 1, 1, 1, 1]
[1, 4, 8, 1, 4, 8, 1, 4, 8]
```

We get the length of on list with *len()*

```
In [33]: print(len(list_2))
         print(len(list_3))

5
9
```

Basic functions of lists are: appending, remove, del, or pop

```
In [34]: list_1.append(10)
         print(list_1)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

We acess the item at a specified index or position of a list with corny paranthesis []. The delete function *del* uses this to remove an item at a specified index.

```
In [35]: del list_1[0]
         print(list_1)

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

The remove function removes the first occurence of the input item in the list, but throws an error if the item is not in the list!

```
In [36]: list_1.remove(10)
         print(list_1)
```

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

```
In [37]: list_1.remove(11)
         print(list_1)
```

```
        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-37-51b7a8c5d187> in <module>
    ----> 1 list_1.remove(11)
          2 print(list_1)


        ValueError: list.remove(x): x not in list
```

pop removes the item at the given position and returns it.

```
In [38]: removed_num = list_1.pop(2)
         print(removed_num)
         print(list_1)
```

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

6.1 List indexing and slicing

To acquire a value at a given index, we need the edgy paranthesis: (don't forget, the index of of the first element is 0!)

```
In [39]: print(list_3)
         print(list_3[0])
         print(list_3[2])
```

```
[1, 4, 8, 1, 4, 8, 1, 4, 8]
1
8
```

Negative index are used to count from the back to the front of the index. Instead of calculating the length of the list, we simpy just use the negative index.

```
In [40]: # get last value in list
         print(list_3[len(list_3)-1])

         # way shorter with -1
         print(list_3[-1])
         print(list_3[-9])
```

```
8
8
1
```

We can also slice lists, meaning we get a portion (a new list) from the list, by smart indexing. It can be complicated, but is super useful to manipulate lists quickly.

```
In [ ]:
```

```
In [41]: # get the first 3 elements
         print(list_1[0:3])

         #shorter, can leave zero away, then is alway from start
         print(list_1[:3])
```

```
[1, 2, 4]
[1, 2, 4]
```

```
In [42]: # get the last 3 elements
         print(list_1[-3:])
```

```
[7, 8, 9]
```

```
In [43]: # get 2nd to 5th element (2, 3, 4, 5)
         # doesn't include the last element!
         print(list_1[2:6])
```

```
[4, 5, 6, 7]
```

Next to start index and end index, we can also define the step size when slicing. This can make a little tricky.

```
In [44]: # get all even number
         print(list_1[::2]) # start and end is not defined, therefore we go over entire list

         # get all odd numbers
         print(list_1[1::2])
```

```
[1, 4, 6, 8]
[2, 5, 7, 9]
```

The sign of the setp also defines the direction we go when slicing

```
In [45]: # print the list in reverse order
         print(list_1[::-1])

         # when going in reverse order
```

```
[9, 8, 7, 6, 5, 4, 2, 1]
```

```
In [46]: # now all together
         print(list_1[-2:2:-3]) # starting at second last index, to 2 index, in reverse order pr
```

```
[8, 5]
```

6.2 Tuples

Tuples are very similar to lists, but with one major difference. You can not change a tuple after creation. This counts for the values in the tuple and also for the tuple size. A tuple is unchangeable.

```
In [47]: x =(1,5,6)
         x

Out[47]: (1, 5, 6)

In [48]: x[0] = 8
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-48-b687365b81a6> in <module>
    ----> 1 x[0] = 8


        TypeError: 'tuple' object does not support item assignment
```

A tuple useful to store multiple values within one variable, or data point.

```
In [49]: jason = (25,195,"dark eyes")
         jason

Out[49]: (25, 195, 'dark eyes')
```

6.3 Loops and Iterators

Most simple way to loop, is to loop until a condition is set. Caution we can get into a endless loop if no end condition is met

```
In [ ]: condition = True
        while(condition):
            print("loop_di_loop") # will run forever
```

Usually we want to loop for a set number of times. Then we use for loops. The `range` functions gives back a (kind of) list of intergers. the `range` object is immutable.

```
In [51]: for i in range(5):
             print(i)
         type(range(5))

0
1
2
3
4


Out[51]: range
```

In the range function we can also specify additionally end and steps

```
In [52]: for i in range(1,10,3):
             print(i)

1
4
7
```

What the for loop really does though, is iterate over every item in list, range object etc (iterable)

```
In [53]: for i in [2, 500, "hello"]:
             print(i)

2
500
hello
```

We can iterate over all variable types that are collections:

```
In [54]: tuple1 = (7,976,80)
         for y in tuple1:
             print(y)
```

```
7
976
80
```

```
In [55]: set1 = set()
         set1.add(1)
         set1.add(8)
```

6.4 Functions

Functions are build with the keyword `def`. After that comes the name of the function, and the input parameters. Different to other commom languages, we dont need to define a return value.

```
In [56]: def my_function():
             print("my first function! Woop woop")
```

```
In [57]: my_function()
```

```
my first function! Woop woop
```

Functions always makes sense, if we know we gonna need code multiple times and only want to write it once. It also helps to make your code modular and to get a better overwiew.

```
In [58]: def greet_person(name):
             if name == "Jason":
                 print('Hello instructor ' + name)
             elif name == 'jason':
                 print('Hello instructor '+ name)
             else:
                 print('Hello student '+ name)
```

```
In [59]: greet_person('Jason')
```

```
Hello instructor Jason
```

```
In [60]: greet_person('zhang')
```

```
Hello student zhang
```

```
In [61]: greet_person('mahmoud')
```

```
Hello student mahmoud
```

Can use functions also in loop

```
In [62]: names = ["zack", "jason", "eva", "phoebe"]

         for i in names:
             greet_person(i)

Hello student zack
Hello instructor jason
Hello student eva
Hello student phoebe
```

We can give input parameters of a function a default value

```
In [63]: def greet_again(person="jason"):

             greet_person(person)

In [64]: greet_again()

Hello instructor jason
```

But we need to check the order of the input parameters. Inputs with default values can not be followed by inputs without default.

```
In [66]: def greet_again_again(person="jason",i):

             greet_person(person)


          File "<ipython-input-66-e670d6ff8a24>", line 1
        def greet_again_again(person="jason",i):
                                         ^
    SyntaxError: non-default argument follows default argument
```

When we use functions, variables declared within the function block only are accessible within the block. In programming this is called the scope.

```
In [ ]: def scope():
            a = 5
            print(a)

In [67]: scope()
         print(a)


        ---------------------------------------------------------------------------
```

```
    NameError                                 Traceback (most recent call last)

    <ipython-input-67-b43cc99a6d02> in <module>
----> 1 scope()
      2 print(a)


    NameError: name 'scope' is not defined
```

If we want to return a value, we simply return it with `return`

```
In [68]: def myfunction():
             return"hello world"
         myfunction()

Out[68]: 'hello world'

In [ ]:
```

6.5 String modification with function calls

With implemented functions of the `string` class, we can modify strings easily.

```
In [ ]:

In [69]: "This,sentence,has,many,commas".split(",") # split a string int substrings

Out[69]: ['This', 'sentence', 'has', 'many', 'commas']

In [70]: " is cool. ".join(["Peter","Zino","Vincent"]) + " is not cool."
         # join a collective of strings together with a desired string

Out[70]: 'Peter is cool. Zino is cool. Vincent is not cool.'

In [71]: "my string, an old string, is good ".find("old string") # find the index of the first o

Out[71]: 14

In [72]: "WHY SO SERIOUS".lower() # convert a string to lower

Out[72]: 'why so serious'

In [73]: "hahaha1".isalpha() # check if string has numerical signs

Out[73]: False

In [74]: "717171".isnumeric()

Out[74]: True

In [75]: ord("A") # get the unicode value of a single character

Out[75]: 65
```