python basics

In [1]:
```
import math
#how do you compute the celling and floor of 2.331?
math.ceil(2.236)
```
Out[1]:
```
3
```

In [2]:
```
math.floor(2.236)
```
Out[2]:
```
2
```

Dynamic typing

In [3]:
```
my_dog=2
my_dog
```
Out[3]:
```
2
```

In [4]:
```
type(my_dog)
```
Out[4]:
```
int
```

Advantage and disadvantage of dynamic typing

Advantage of dynamic typing

*easy to work with

*less development time

Disadvantage of Dynamic typing

*unexpected bugs!

*you need to be aware of type()

In [8]:
```
# index positioned to insert objects
print("the {2} {1} {0}".format("bar","strong","quick"))
```
```
thequick strong bar
```

In [9]:
```
print("key 1 : {a} ,key 2 :{b},key 3 :{c}".format(a=5,b="hello",c=300))
```
```
key 1 : 5 ,key 2 :hello,key 3 :300
```

Q1. write a python program that return the first and

last character of any gien word

In [10]:
```
def first_last_character(word):
    return word[0]+word[-1]
```

In [11]:
```
first_last_character("stallone")
```

Out[11]:
```
'se'
```

Q2.write a python program to replace any "_","." and "@" with space character

In [12]:
```
def replace_character_in(string):
    return string.replace("_"," ").replace("."," ").replace("@", " ")
```

In [14]:
```
replace_character_in("write @ a_python, program_to@ replace_character")
```

Out[14]:
```
'write   a python, program to  replace character'
```

whitesapce is important

In [1]:
```
listofnumbers=[1,2,3,4,5,6]
for number in listofnumbers:
    print(number),
    if(number % 2==0):
        print("is even")
    else:
        print("is odd")
print("all done")
```

```
1
is odd
2
is even
3
is odd
4
is even
5
is odd
6
is even
all done
```

In [2]:
```
import numpy as np
A=np.random.normal(25.0,5.0,10)
print(A)
```

```
[31.23811858 29.40241719 23.37569672 17.86474138 24.68260954 24.06054678
 27.89300322 19.18938569 22.4159282  28.27568768]
```

```
In [3]: x=[1,2,3,4,5,6]
        print(len(x))

        6
```

post colon

```
In [4]: x[3:]

Out[4]: [4, 5, 6]
```

pre colon

```
In [5]: x[:4]

Out[5]: [1, 2, 3, 4]
```

negative syntax

```
In [6]: x[-2:]

Out[6]: [5, 6]
```

adding list to list

You can also change lists around. Let's say I want to add a list to the list. I can use the extend function for that, as shown in the following code block:

```
In [7]: x.extend([7,8])
        x

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

The append function

If you want to just add one more thing to that list, you can use the append function. So I just want to stick the number 9 at the end, there we go:

```
In [8]: x.append(9)

In [9]: x

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

In [10]:
```
y=[10,11,12]
listoflists=[x,y]
listoflists
```
Out[10]:
```
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [10, 11, 12]]
```

In [11]:
```
y[1]
```
Out[11]:
```
11
```

The sort function

Finally, let's have a built-in sort function that you can use:

In [12]:
```
z=[3,2,1]
z.sort()
z
```
Out[12]:
```
[1, 2, 3]
```

Reverse sort

In [13]:
```
z.sort(reverse=True)
z
```
Out[13]:
```
[3, 2, 1]
```

Tuples

Tuples are just like lists, except they're immutable, so you can't actually extend, append, or sort them. They are what they are, and they behave just like lists, apart from the fact that you can't change them, and you indicate that they are immutable and are tuple, as opposed to a list, using parentheses instead of a square bracket. So you can see they work pretty much the same way otherwise:

In [14]:
```
#tuples are just immutable list.use() instead of []
x=(1,2,3)
len(x)
```
Out[14]:
```
3
```

Dereferencing an element

We can also dereference the elements of a tuple, so element number 2 again would be the third element, because we start counting from 0, and that will give me back the number □

```
In [15]: y=(4,5,6)
```

```
In [16]: y[2]
```

```
Out[16]: 6
```

List of tuples

We can also, like we could with lists, use tuples as elements of a list.

```
In [17]: listoftuples=[x,y]
         listoftuples
```

```
Out[17]: [(1, 2, 3), (4, 5, 6)]
```

```
In [18]: (age,income)="32,120000".split(",")
         print(age)
         print(income)
```

```
32
120000
```

Dictionaries

Finally, the last data structure that we'll see a lot in Python is a dictionary, and you can think of that as a map or a hash table in other languages. It's a way to basically have a sort of mini-database, sort of a key/value data store that's built into Python. So let's say, I want to build up a little dictionary of Star Trek ships and their captains

like a map or hash table in other language in tother languages

```
In [19]: captains={}
         captains["Enterprise"]="Kirk"
         captains["Enterprise D"]="picard"
         captains["Deep space nine"]="Sisko"
         captains["voyager"]="janeway"
         print(captains["voyager"])
```

```
janeway
```

```
In [20]: print(captains.get("Enterprise"))
```

```
Kirk
```

```
In [21]: print(captains.get("NX-01"))
```

```
None
```

Iterating through entries

In [22]:
```
for ship in captains:
    print(ship+": " +captains[ship])
```

```
Enterprise: Kirk
Enterprise D: picard
Deep space nine: Sisko
voyager: janeway
```

function in python

you repeat a set of operations over and over again with different parameters.

In [25]:
```
def squareit(x):
    return x*x
print(squareit(2))
```

```
4
```

In [29]:
```
#you can pass functions around as parameters
def DoSomething(f,x):
    return f(x)
print(DoSomething(squareit,3))
```

```
9
```

Lambda functions - functional programming

One more thing that's kind of a Python-ish sort of a thing to do, which you might not see in other languages is the concept of lambda functions, and it's kind of called functional programming. The idea is that you can include a simple function into a function. This makes the most sense with an example:

In [32]:
```
#lambda functions let you simple functions
print(DoSomething(lambda x:x*x*x,3))
```

```
27
```

Understanding boolean expressions

Boolean expression syntax is a little bit weird or unusual, at least in Python:

In [33]:
```
print(1==2)
```

```
False
```

In [35]:
```
print(True or False)
```

```
True
```

the if statement

In [36]:
```
print(1 is 3)
```

```
False

<>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
C:\Users\satya shukla\AppData\Local\Temp\ipykernel_2440\453973788.py:1: Sy
ntaxWarning: "is" with a literal. Did you mean "=="?
  print(1 is 3)
```

the if- else loop

In [37]:
```
if 1 is 3:
    print("how did that happen?")
elif 1>3:
    print("yikes")
else:
    print("all is well with the world")
```

```
all is well with the world

<>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
C:\Users\satya shukla\AppData\Local\Temp\ipykernel_2440\606507094.py:1: Sy
ntaxWarning: "is" with a literal. Did you mean "=="?
  if 1 is 3:
```

Looping

The last concept I want to cover in our Python basics is looping, and we saw a couple of this already,but let's just do another one:

In [38]:
```
for x in range(10):
    print(x)
```

```
0
1
2
3
4
5
6
7
8
9
```

In [42]:
```
for x in range(10):
    if(x is 1):
        continue
    if(x>5):
        break
    print(x),
```

```
0
2
3
4
5

<>:2: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:2: SyntaxWarning: "is" with a literal. Did you mean "=="?
C:\Users\satya shukla\AppData\Local\Temp\ipykernel_2440\104298138.py:2: Sy
ntaxWarning: "is" with a literal. Did you mean "=="?
  if(x is 1):
```

The while loop

Another syntax is the while loop. This is kind of a standard looping syntax that you see in most languages:

In [43]:
```
x=0
while(x<10):
    print(x),
    x+=1
```

```
0
1
2
3
4
5
6
7
8
9
```

In [48]:

```
x=[2,3,4,5,6,7,8,9]
for number in (x):
    print(number),
    if(number % 2==0):
        print("is even")
    else:
        print("is odd")
print("hooray !we are all done. lets party !")
```

```
2
is even
3
is odd
4
is even
5
is odd
6
is even
7
is odd
8
is even
9
is odd
hooray !we are all done. lets party !
```

In [ ]: