Basics of Python

Python is a high-level, dynamically typed multiparadigm programming language. Python code is often said to be almost like pseudocode, since it allows you to express very powerful ideas in very few lines of code while being very readable.

Fundamental types

Integers

Integer literals are created by any number without a decimal or complex component.

In [1]:

```
# integers
a = 7
print("fggdgsgd" + str(a))
```

```
fggdgsgd7
```

In [ ]:

```
```

Floats

Float literals can be created by adding a decimal component to a number.

In [2]:

```
# float
x = 1.0
type(x)
```

Out[2]:

```
float
```

Boolean

Boolean can be defined by typing True/False without quotes

In [3]:

```
# boolean
b1 = True
b2 = False

type(b1)
```

Out[3]:

```
bool
```

Strings

String literals can be defined with any of single quotes ('), double quotes (") or triple quotes (''' or """). All give the same result with two important differences.

If you quote with single quotes, you do not have to escape double quotes and vice-versa. If you quote with triple quotes, your string can span multiple lines.

In [4]:

```
# string
name1 = 'your name vvgafdasfasfasfa'
type(name1)
```

Out[4]:

```
str
```

Complex

Complex literals can be created by using the notation x + yj where x is the real component and y is the imaginary component.

In [5]:

```
#complex numbers: note the use of `j` to specify the imaginary part

j = 1.0 - 2.0j
print(type(j))
j
```

```
<class 'complex'>
```

Out[5]:

```
(1-2j)
```

In [6]:

```
print(j.real, j.imag)
```

```
1.0 -2.0
```

Variables

Defining

A variable in Python is defined through assignment. There is no concept of declaring a variable outside of that assignment.

In [7]:

```
tenth = 10
tenth
```

Out[7]:

```
10
```

Strong Typing

While Python allows you to be very flexible with your types, you must still be aware of what those types are. Certain operations will require certain types as arguments.

In [8]:

```
'Day ' +str(1)
```

Out[8]:

```
'Day 1'
```

In [9]:

```
#'Day ' + 1
```

Simple Expressions

Boolean Evaluation

Boolean expressions are created with the keywords and, or, not and is. For example:

In [10]:

```
True and True
```

Out[10]:

```
True
```

In [11]:

```
True or False
```

Out[11]:

```
True
```

In [12]:

```
not True
```

Out[12]:

```
False
```

In [13]:

```
not False
```

Out[13]:

```
True
```

In [14]:

```
True is True
```

Out[14]:

```
True
```

In [15]:

```
True is False
```

Out[15]:

```
False
```

In [16]:

```
'a' is 'a'
```

Out[16]:

```
True
```

Branching (if / elif / else)

Python provides the if statement to allow branching based on conditions. Multiple elif checks can also be performed followed by an optional else clause. The if statement can be used with any evaluation of truthiness.

In [17]:

```
i = 1
if (i < 3):
    print('less than 3')
    if i<2:
        print("second condition")
elif i < 5:
    print('less than 5')
else:
    print('5 or more')
```

```
less than 3
second condition
```

In [18]:

```
num = int(input("enter any number: "))
if num >= 0:
    print("number is positive")
else:
    print("number is negative")
```

```
enter any number: 0
number is positive
```

Block Structure and Whitespace

The code that is executed when a specific condition is met is defined in a "block." In Python, the block structure is signalled by changes in indentation. Each line of code in a certain block level must be indented equally and indented more than the surrounding scope. The standard (defined in PEP-8) is to use 4 spaces for each level of block indentation. Statements preceding blocks generally end with a colon (:).

Because there are no semi-colons or other end-of-line indicators in Python, breaking lines of code requires either a continuation character (\ as the last char) or for the break to occur inside an unfinished structure (such as open parentheses).

Advanced Types: Containers

One of the great advantages of Python as a programming language is the ease with which it allows you to manipulate containers. Containers (or collections) are an integral part of the language and, as you'll see, built in to the core of the language's syntax. As a result, thinking in a Pythonic manner means thinking about containers.

Lists

The first container type that we will look at is the list. A list represents an ordered, mutable collection of objects. You can mix and match any type of object in a list, add to it and remove from it at will.

Creating Empty Lists. To create an empty list, you can use empty square brackets or use the list() function with no arguments.

In [19]:

```
l = []
l
```

Out[19]:

```
[]
```

In [20]:

```
l = list()
l
```

Out[20]:

```
[]
```

Initializing Lists. You can initialize a list with content of any sort using the same square bracket notation. The list() function also takes an iterable as a single argument and returns a shallow copy of that iterable as a new list. A list is one such iterable as we'll see soon, and we'll see others later.

In [21]:

```
l1 = ["dsad", 'b', 'c']
l2 = ['a',6,4.0]
```

In [22]:

```
l1  + l2
```

Out[22]:

```
['dsad', 'b', 'c', 'a', 6, 4.0]
```

A Python string is also a sequence of characters and can be treated as an iterable over those characters. Combined with the list() function, a new list of the characters can easily be generated.

In [23]:

```
l=list("jkdfgkdgj")
l
```

Out[23]:

```
['j', 'k', 'd', 'f', 'g', 'k', 'd', 'g', 'j']
```

Adding. You can append to a list very easily (add to the end) or insert at an arbitrary index.

In [24]:

```
for i in l:
    print(i)
```

```
j
k
d
f
g
k
d
g
j
```

Loops

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

For loop

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

In [25]:

```
# Program to find the sum of all numbers stored in a list
# Program to find the sum of all numbers stored in a list

# List of numbers
numbers = ["Fsdfsf","fsdfs","sdffsfs"]

# variable to store the sum
summ=''

# iterate over the list
for x in numbers:
    summ = summ+x

# Output: The sum is 48
print(summ)
```

```
Fsdfsffsdfssdffsfs
```

In [ ]:

```

```

The range() function

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).

We can also define the start, stop and step size as range(start,stop,step size). step size defaults to 1 if not provided.

This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

To force this function to output all the items, we can use the function list().

The following example will clarify this.

In [26]:

```
range(6)
```

Out[26]:

```
range(0, 6)
```

In [27]:

```
list(range(10))
```

Out[27]:

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

In [28]:

```
print(list(range(8, 2,-2)))
```

```
[8, 6, 4]
```

In [29]:

```
print(list(range(1000,10,-200)))
```

```
[1000, 800, 600, 400, 200]
```

In [30]:

```
print(list(range(2, 20, 5)))
```

```
[2, 7, 12, 17]
```

We can use the range() function in for loops to iterate through a sequence of numbers. It can be combined with

We can use the range() function in for loops to iterate through a sequence of numbers. It can be combined with the len() function to iterate though a sequence using indexing. Here is an example.

In [31]:

```
# Program to iterate through a list using indexing

lis = ['vnk', 'mnk', 'gpg','nitk']

# iterate over the list using index
for i in range(len(lis)-1):
    print("I like", lis[i])
```

```
I like vnk
I like mnk
I like gpg
```

In [32]:

```
a=[1,2,3,4,5,6,7,8]
for i in a:
    print("hey its : {}".format(i))
```

```
hey its : 1
hey its : 2
hey its : 3
hey its : 4
hey its : 5
hey its : 6
hey its : 7
hey its : 8
```

break and continue statement

In Python, break and continue statements can alter the flow of a normal loop.

Loops iterate over a block of code until test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without cheking test expression.

The break and continue statements are used in these cases.

break

The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

In [33]:

```
num =[1,2,3,4,5,6,7,8,9,10]
for i in num:
    if (i== 5) or (i==9):
        continue
    print(i)
```

```
1
2
3
4
6
7
8
10
```

In [34]:

```
num =[1,2,3,4,5,6,7,8,9,10]
for i in num:
    if (i== 5):
        continue
    print(i)
```

```
1
2
3
4
6
7
8
9
10
```

In [35]:

```
num =[1,2,3,4,5,6,7,8,9,10]
for i in num:
    if i== 5:
        print('five level :',i)
        break
    print(i)
```

```
1
2
3
4
five level : 5
```

In this program, we iterate through the "string" sequence. We check if the letter is "i", upon which we break from the loop. Hence, we see in our output that all the letters up till "i" gets printed. After that, the loop terminates.

continue

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

In [36]:

```
# Program to show the use of continue statement inside loops

for val in "string":
    if val == "i":
        continue
    print(val)

print("The end")
```

```
s
t
r
n
g
The end
```

This program is same as the above example except the break statement has been replaced with continue.

We continue with the loop, if the string is "i", not executing the rest of the block. Hence, we see in our output that all the letters except "i" gets printed.

Strings

Strings are used to record the text information such as name. In Python, Strings act as “Sequence” which means Python tracks every element in the String as a sequence. This is one of the important features of the Python language. For example, Python understands the string "hello' to be a sequence of letters in a specific order which means the indexing technique to grab particular letters (like first letter or the last letter).

Creating a String

In Python, either single quote (‘) or double quotes (“) must be used while creating a string.

```
For example:
```

In [37]:

```
# Single word
'hello'
```

Out[37]:

```
'hello'
```

In [38]:

```
# Entire phrase
'This is also a string'
```

Out[38]:

```
'This is also a string'
```

In [39]:

```
# We can also use double quote
"String built with double quotes"
```

Out[39]:

```
'String built with double quotes'
```

In [40]:

```
# Be careful with quotes!
#' I'm using single quotes, but will create an error'
```

The above code results in an error as the text “I’m” stops the string. Here, a combination of single quotes and double quotes can be used to get the complete statement.

In [41]:

```
"Now I'm ready to use the single quotes inside a string!"
```

Out[41]:

```
"Now I'm ready to use the single quotes inside a string!"
```

Printing a String

We can automatically display the output strings using Jupyter notebook with just a string in a cell. But,the correct way to display strings in your output is by using a print function.

In [42]:

```
# We can simply declare a string
'Hello World'
```

Out[42]:

```
'Hello World'
```

In [43]:

```
# note that we can't output multiple strings this way
'Hello World 1'
```

```
'Hello World 2'
```

Out[43]:

```
'Hello World 2'
```

In Python 2, the output of the below code snippet is displayed using "print" statement as shown in the below syntax but the same syntax will throw error in Python 3.

In [44]:

```
#print 'Hello World 1'
#print 'Hello World 2'
#print 'Use \n to print a new line'
#print '\n'
#print 'See what I mean?'
```

Python 3 Alert!

Note that, In Python 3, print is a function and not a statement. So you would print statements like this: print('Hello World')

If you want to use this functionality in Python2, you can import form the future module.

Caution: After importing this; you won't be able to choose the print statement method anymore. So pick the right one whichever you prefer depending on your Python installation and continue on with it.

In [45]:

```
# To use print function from Python 3 in Python 2
from __future__ import print_function

print('Hello World')
```

```
Hello World
```

String Basics

In Strings, the length of the string can be found out by using a function called len().

In [46]:

```
len('Hello World')
```

Out[46]:

```
11
```

String Indexing

We know strings are a sequence, which means Python can use indexes to call all the sequence parts. Let's learn how String Indexing works. • We use brackets [] after an object to call its index. • We should also note that indexing starts at 0 for Python. Now, Let's create a new object called s and the walk through a few examples of indexing.

In [47]:

```
# Assign s as a string
s = 'Hello World'
```

In [48]:

```
#Check
s
```

Out[48]:

```
'Hello World'
```

In [49]:

```
# Print the object
print(s)
```

```
Hello World
```

Let's start indexing!

In [50]:

```
# Show first element (in this case a letter)
s[0]
```

Out[50]:

```
'H'
```

In [51]:

```
s[1]
```

Out[51]:

```
'e'
```

In [52]:

```
s[2]
```

Out[52]:

```
'l'
```

We can use a : to perform *slicing* which grabs everything up to a designated point. For example:

In [53]:

```
# Grab everything past the first term all the way to the length of s which is len(s)
s[1:]
```

Out[53]:

```
'ello World'
```

In [54]:

```
# Note that there is no change to the original s
s
```

Out[54]:

```
'Hello World'
```

In [55]:

```
# Grab everything UP TO the 3rd index
s[:3]
```

Out[55]:

```
'Hel'
```

Note the above slicing. Here we're telling Python to grab everything from 0 up to 3. It doesn't include the 3rd index. You'll notice this a lot in Python, where statements and are usually in the context of "up to, but not including".

In [56]:

```
#Everything
s[:]
```

Out[56]:

```
'Hello World'
```

We can also use negative indexing to go backwards.

In [57]:

```
# Last letter (one index behind 0 so it loops back around)
s[-1]
```

Out[57]:

```
'd'
```

In [58]:

```
# Grab everything but the last letter
s[:-1]
```

Out[58]:

```
'Hello Worl'
```

Index and slice notation is used to grab elements of a sequenec by a specified step size (where in 1 is the default size). For instance we can use two colons in a row and then a number specifying the frequency to grab elements. For example:

In [59]:

```
# Grab everything, but go in steps size of 1
s[::1]
```

Out[59]:

```
'Hello World'
```

In [60]:

```
# Grab everything, but go in step sizes of 2
s[::2]
```

Out[60]:

```
'HloWrd'
```

In [61]:

```
# We can use this to print a string backwards
s[::-1]
```

Out[61]:

```
'dlroW olleH'
```

String Properties

Immutability is one the finest string property whichh is created once and the elements within it cannot be changed or replaced. For example:

In [62]:

```
s
```

Out[62]:

```
'Hello World'
```

```
hello world
```

In [63]:

```
# Let's try to change the first letter to 'x'
s[0] = 'x'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-63-976942677f11> in <module>
      1 # Let's try to change the first letter to 'x'
----> 2 s[0] = 'x'

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

In [ ]:

```
# Concatenate strings!
s + ' concatenate me!'
```

In [ ]:

```
s
```

In [ ]:

```
# We can reassign s completely though!
s = s + ' concatenate me!'
```

In [ ]:

```
s
```

In [ ]:

```
#We can use the multiplication symbol to create repetition!
letter = 'z'
letter*10
```

Basic Built-in String methods

In Python, Objects have built-in methods which means these methods are functions present inside the object (we will learn about these in much more depth later) that can perform actions or commands on the object itself.

Methods can be called with a period followed by the method name. Methods are in the form:

object.method(parameters)

Where parameters are extra arguments which are passed into the method. Right now, it is not necessary to make 100% sense but going forward we will create our own objects and functions.

Here are some examples of built-in methods in strings:

In [ ]:

```
s
```

In [ ]:

```
# Upper Case a string
s.upper()
```

In [ ]:

```
# Lower case
s.lower()
```

In [ ]:

```
# Split a string by blank space (this is the default)
s.split()
```

In [ ]:

```
# Split by a specific element (doesn't include the element that was split on)
s.split('W')
```

There are many more methods than the ones covered here. To know more about the String functions, Visit the advanced String section.

Print Formatting

Print Formatting ".format()" method is used to add formatted objects to the printed string statements.

Let's see an example to clearly understand the concept.

'Insert another string with curly brackets: {}'.format('The inserted string')

Location and Counting

In [ ]:

```
#Num of times it is repeated

s.count('W')
```

In [ ]:

```
#index Location
print(s.find('W'))

print(s.find('H'))
```

Formatting

The center() method allows you to place your string 'centered' between a provided string with a certain length.

In [ ]:

```
s.center(50,'z')
```

expandtabs() will expand tab notations \t into spaces. Let's see an example to understand the concept.

In [ ]:

```
'hello\thi'.expandtabs()
```

In [ ]:

```
print('hello\thi')
```

is check methods

These various methods below check it the string is some case. Lets explore them:

In [ ]:

```
s = 'hello'
```

isalpha() wil return "True" if all characters in S are letters only.

In [ ]:

```
s.isalpha()
```

In [ ]:

```
ss ='abcxjA1'
print(ss.isalpha())
```

In [ ]:

```
s.islower()
```

In [ ]:

```
ss.islower()
```

isspace() will return "True" if all characters in S are whitespace.

In [ ]:

```
s1 = '   '
s1.isspace()
```

In [ ]:

```
s2 = "hello"
s2.istitle()
```

isupper() will return "True" if all cased characters in S are uppercase and there is at least one cased character in S, False otherwise.

In [ ]:

```
s.isupper()
```

Another method is endswith() which is essentially same as a boolean check on s[-1]

In [ ]:

```
s
```

In [ ]:

```
s.endswith('o')
```

Built-in Reg. Expressions

In Strings, there are some built-in methods which is similar to regular expression operations. • Split() function is used to split the string at a certain element and return a list of the result. • Partition is used to return a tuple that includes the separator (the first occurrence), the first half and the end half.

In [ ]:

```
s.split('e')
```

In [ ]:

```
s.partition('e')
```

Lists

Earlier, while discussing introduction to strings we have introduced the concept of a *sequence* in Python. In

Python, Lists can be considered as the most general version of a "sequence". Unlike strings, they are mutable which means the elements inside a list can be changed!

Lists are constructed with brackets [] and commas separating every element in the list.

Let's go ahead and see how we can construct lists!

In [ ]:

```
# Assign a list to an variable named my_list
my_list = [1,2,3]
```

We just created a list of integers, but lists can actually hold different object types. For example:

In [ ]:

```
my_list = ['A string',23,100.232,'o']
```

Just like strings, the len() function will tell you how many items are in the sequence of the list.

In [ ]:

```
len(my_list)
```

Indexing and Slicing

Indexing and slicing of lists works just like in Strings. Let's make a new list to remind ourselves of how this works:

In [ ]:

```
my_list = ['one','two','three',4,5]
```

In [ ]:

```
# Grab element at index 0
my_list[0]
```

In [ ]:

```
# Grab index 1 and everything past it
my_list[1:]
```

In [ ]:

```
# Grab everything UP TO index 3
my_list[:3]
```

We can also use "+" to concatenate lists, just like we did for Strings.

In [ ]:

```
my_list + ['new item',5]
```

Note: This doesn't actually change the original list!

In [ ]:

```
my_list
```

In this case, you have to reassign the list to make the permanent change.

In [ ]:

```
# Reassign
```

```
my_list = my_list + ['add new item permanently']
```

In [ ]:

```
my_list
```

We can also use the * for a duplication method similar to strings:

In [ ]:

```
# Make the list double
my_list * 2
```

In [ ]:

```
# Again doubling not permanent
my_list
```

Basic List Methods

If you are familiar with another programming language, start to draw parallels between lists in Python and arrays in other language. There are two reasons which tells why the lists in Python are more flexible than arrays in other programming language:

a. They have no fixed size (which means we need not to specify how big the list will be) b. They have no fixed type constraint

Let's go ahead and explore some more special methods for lists:

In [ ]:

```
# Create a new list
l = [1,2,3]
```

Use the append method to permanently add an item to the end of a list:

In [ ]:

```
# Append
l.append('append me!')
```

In [ ]:

```
l
```

Use pop to "pop off" an item from the list. By default pop takes off the last index, but you can also specify which index to pop off. Let's see an example:

In [ ]:

```
# Pop off the 0 indexed item
l.pop(0)
```

In [ ]:

```
l
```

In [ ]:

```
# Assign the popped element, remember default popped index is -1
popped_item = l.pop()
```

In [ ]:

```
popped_item
```

In [ ]:

```
l
```

Note that lists indexing will return an error if there is no element at that index. For example:

In [ ]:

```
l[100]
```

We can use the sort method and the reverse methods to also effect your lists:

In [ ]:

```
new_list = ['a','e','x','b','c']
```

In [ ]:

```
#Show
new_list
```

In [ ]:

```
# Use reverse to reverse order (this is permanent!)
new_list.reverse()
```

In [ ]:

```
new_list
```

In [ ]:

```
# Use sort to sort the list (in this case alphabetical order, but for numbers it will go ascending)
new_list.sort()
```

In [ ]:

```
new_list
```

Nesting Lists

Nesting Lists is one of the great features in Python data structures. Nesting Lists means we can have data structures within data structures.

For example: A list inside a list.

Let's see how Nesting lists works!

In [ ]:

```
# Let's make three lists
lst_1=[1,2,3]
lst_2=[4,5,6]
lst_3=[7,8,9]

# Make a list of lists to form a matrix
matrix = [lst_1,lst_2,lst_3]
```

In [ ]:

```
# Show
matrix
```

In [ ]:

```
matrix[2][1]
```

We can re-use indexing to grab elements, but now there are two levels for the index.

a. The items in the matrix object b. The items inside the list

In [ ]:

```python
# Grab first item in matrix object
matrix[0]
```

In [ ]:

```python
# Grab first item of the first item in the matrix object
matrix[0][0]
```

List Comprehensions

Python has an advanced feature called list comprehensions which allows for quick construction of lists.

Before we try to understand list comprehensions completely we need to understand "for" loops.

So don't worry if you don't completely understand this section, and feel free to just skip it since we will return to this topic later.

Here are few of oue examples which helps you to understand list comprehensions.

In [ ]:

```python
matrix
```

In [ ]:

```python
for row in matrix:
    print(row)
```

In [ ]:

```python
# Build a list comprehension by deconstructing a for loop within a []
first_col = [row[0] for row in matrix]
```

In [ ]:

```python
first_col
```

In [ ]:

```python
#conditional list Comprehension
number_list = [ x for x in range(20) if x % 2 == 0]
print(number_list)
```

In [ ]:

```python
for x in range(20):
    if x % 2 == 0:
        print(x)
```

In [ ]:

```python
#check two conditions is x divisible by 2 and 5 in range of 20
num = []
for x in range(20):
    if x % 2 == 0:
        if x % 5 == 0:
            num.append(x)
print(num)
```

In [ ]:

```
[x for x in range(20) if x % 2 == 0 if x % 5 == 0]
```

Advanced Lists

In this series of lectures, we will be diving a little deeper into all the available methods in a list object. These are just methods that should encountered without some additional exploring. Its pretty likely that you've already encountered some of these yourself!

Lets begin!

In [ ]:

```
l = [1,2,3]
```

append

Definitely, You have used this method by now, which merely appends an element to the end of a list:

In [ ]:

```
l.append(4)

l
```

count

We discussed this during the methods lectures, but here it is again. count() takes in an element and returns the number of times it occures in your list:

In [ ]:

```
l.count(10)
```

In [ ]:

```
l.count(3)
```

extend

Many times people find the difference between extend and append to be unclear. So note that,

append: Appends object at end

In [ ]:

```
x = [1, 2, 3]
x.append([4, 5])
print(x)
```

extend: extends list by appending elements from the iterable

In [ ]:

```
x = [1, 2, 3]
x.extend('ty')
print(x)
```

Note how extend append each element in that iterable. That is the key difference.

index

index returns the element placed as an argument. Make a note that if the element is not in the list then it returns an error.

In [ ]:

```
l.index(4)
```

In [ ]:

```
l
```

insert

Two arguments can be placed in insert method.

Syntax: insert(index,object)

This method places the object at the index supplied. For example:

In [ ]:

```
# Place a letter at the index 2
l.insert(2,'inserted')
```

In [ ]:

```
l
```

remove

The remove() method removes the first occurrence of a value. For example:

In [ ]:

```
l
```

In [ ]:

```
l.remove('inserted')
```

In [ ]:

```
l
```

In [ ]:

```
l = [1,2,3,4,3]
```

In [ ]:

```
l.remove(3)
```

In [ ]:

```
l
```

reverse

As the name suggests, reverse() helps you to reverse a list. Note this occurs in place! Meaning it effects your list permanently.

In [ ]:

```
l
```

In [ ]:

```
l.reverse()
```

In [ ]:

```
l
```

sort

sort will sort your list in place:

In [ ]:

```
l
```

In [ ]:

```
l.sort()
```

In [ ]:

```
l
```

In [ ]:

```
l.sort(reverse = True)
```

In [ ]:

```
l
```

In [ ]:

```

```