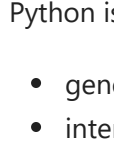


Getting Started with Python

About Python



Python is a

- general purpose programming language
- interpreted, not compiled
- both **dynamically typed** and **strongly typed**
- supports multiple programming paradigms: object oriented, functional
- comes in 2 main versions in use today: 2.7 and 3.x

Why Python for Data Science?

Python is great for data science because:

- general purpose programming language (as opposed to R)
- faster idea to execution to deployment
- battle-tested
- mature ML libraries

And it is easy to learn!



Python's Interactive Console : The Interpreter

- The Python interpreter is a console that allows interactive development
- We are currently using the Jupyter notebook, which uses an advanced Python interpreter called IPython
- This gives us much more power and flexibility

Let's try it out!

```
In [1]: print("Hello World!") #As usual with any language we start with with the print function

Hello World!
```

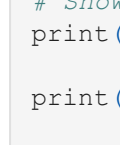
What are we going to learn today?

- CHAPTER 1 - Python Basics
 - **Strings**
 - Creating a String, variable assignments
 - String Indexing & Slicing
 - String Concatenation & Repetition
 - Basic Built-in String Methods
 - **Numbers**
 - Types of Numbers
 - Basic Arithmetic
- CHAPTER 2 - Data Types & Data Structures
 - Lists
 - Dictionaries
 - Sets & Booleans
- CHAPTER 3 - Python Programming Constructs
 - Loops & Iterative Statements
 - if,elif,else statements
 - for loops, while loops
 - Comprehensions
 - Exception Handling
 - Modules, Packages,
 - File I/O operations

CHAPTER - 1 : Python Basics

Let's understand

- Basic data types
- Variables and Scoping
- Modules, Packages and the **import** statement
- Operators



Concept-Alert

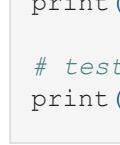
Strings

Strings are used in Python to record text information, such as name. Strings in Python are actually a sequence, which basically means Python keeps track of every element in the string as a sequence. For example, Python understands the string "hello" to be a sequence of letters in a specific order. This means we will be able to use indexing to grab particular letters (like the first letter, or the last letter).

This idea of a sequence is an important one in Python and we will touch upon it later on in the future.

In this lecture we'll learn about the following:

- 1.) Creating Strings
- 2.) Printing Strings
- 3.) String Indexing and Slicing
- 4.) String Properties
- 5.) String Methods
- 6.) Print Formatting



Technical-Alert

Creating a String

To create a string in Python you need to use either single quotes or double quotes. For example:

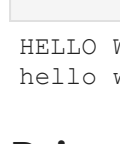
```
In [2]: # Single word
print("hello")

print() # Used to have a line space between two sentences. Try deleting this line & seeing the difference.

# Entire phrase
print("This is also a string")
```

```
hello

This is also a string
```



Concept-Alert

Variables : Store your Value in me!

In the code below we begin to explore how we can use a variable to which a string can be assigned. This can be extremely useful in many cases, where you can call the variable instead of typing the string everytime. This not only makes our code clean but it also makes it less redundant. Example syntax to assign a value or expression to a variable.

variable_name = value or expression

Now let's get coding!! With the below block of code showing how to assign a string to variable.

```
In [3]: s = 'New York'

print(s)

print(type(s))

print(len(s)) # what's the string length
```

```
New York
<class 'str'>
6
```



Technical-Alert

String Indexing

We know strings are a sequence, which means Python can use indexes to call parts of the sequence. Let's learn how this works.

In Python, we use brackets [] after an object to call its index. We should also note that indexing starts at 0 for Python. Let's create a new object called **s** and the walk through a few examples of indexing.

```
In [4]: # Assign s as a string
s = "Hello World"
```

```
In [5]: # Print the object
print(s)

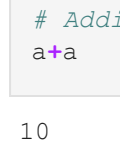
print()

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

print()

# Show the second element (also a letter)
print(s[1])

Hello World
H
```



Technical-Alert

String Concatenation and Repetition

String Concatenation is a process to combine two strings. It is done using the '+' operator.

String Repetition is a process of repeating a same string multiple times

The examples of the above concepts is as follows.

```
In [6]: # concatenation (addition)

s1 = "Hello"
s2 = "World"
print(s1 + " " + s2)
```

```
Hello World
```

```
In [7]: # repetition (multiplication)

print("Hello." * 3)
print("-" * 10)
```

```
Hello_Hello_Hello_
-----
          
```



Technical-Alert

String Slicing & Indexing

String Indexing is used to select the letter at a particular index/position.

String Slicing is a process to select a subset of an entire string

The examples of the above stated are as follows

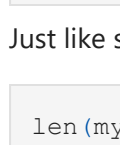
```
In [8]: s = "Namaste World"

# print sub strings
print(s[1]) #this is indexing.
print(s[6:11]) #this is known as slicing.
print(s[-5:-1])

# test substring membership
print("Wor" in s)
```

```
a
e Wor
Worl
True
```

Note the above slicing. Here we're telling Python to grab everything from 6 up to 10 and from fifth last to second last. You'll notice this a lot in Python, where statements and are usually in the context of 'up to, but not including'.



Technical-Alert

Basic Built-in String methods

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

We call methods with a period and then the method name. Methods are in the form:

object.method(parameters)

Where parameters are extra arguments we can pass into the method. Don't worry if the details don't make 100% sense right now. Later on we will be creating our own objects and functions!

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

```
In [9]: s = "Hello World"

print(s.upper()) ## Convert all the element of the string to Upper case.!!
print(s.lower()) ## Convert all the element of the string to Lower case.!!

HELLO WORLD
hello world
```

Print Formatting

We can use the .format() method to add formatted objects to printed string statements.

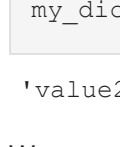
The easiest way to show this is through an example:

```
In [10]: name = "Einstein"
age = 22
married = True

print("My name is {}, my age is {}, and it is {} that I am married".format(name, age, married))

print("My name is {}, my age is {}, and it is {} that I am married".format(name, age, married))

My name is Einstein, my age is 22, and it is True that I am married
My name is Einstein, my age is 22, and it is True that I am married
```

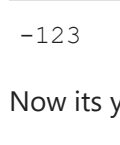


Concept-Alert

Numbers

Having worked with string we will turn our attention to numbers We'll learn about the following topics:

- 1.) Types of Numbers in Python
- 2.) Basic Arithmetic
- 3.) Object Assignment in Python



Concept-Alert

Types of numbers

Python has various "types" of numbers (numeric literals). We'll mainly focus on integers and floating point numbers.

Integers are just whole numbers, positive or negative. For example: 2 and -2 are examples of integers.

Floating point numbers in Python are notable because they have a decimal point in them, or use an exponential (e) to define the number. For example 1.20 and -1.2 are examples of floating point numbers. 4E2 (4 times 10 to the power of 2) is also an example of a floating point number in Python.

Throughout this course we will be mainly working with integers or simple float number types.

Here is a table of the two main types we will spend most of our time working with some examples:

Examples	Number "Type"
12,-5,1000	Integers
12.05,2e2,3E2	Floating-point numbers

Now let's start with some basic arithmetic.

Basic Arithmetic

```
In [11]: # Addition
print(2+1)

# Subtraction
print(2-1)

# Multiplication
print(2*2)

# Division
print(3/2)
```

```
3
1
4
1.5
```

Arithmetic continued

```
In [12]: # Powers
2**3
```

```
Out[12]: 8
```

```
In [13]: # Order of Operations followed in Python
2 + 10 * 10 ** 3
```

```
Out[13]: 105
```

```
In [14]: # Can use parenthesis to specify orders
(2+10) * (10**3)
```

```
Out[14]: 156
```



Technical-Alert

Variable Assignments

Now that we've seen how to use numbers in Python as a calculator let's see how we can assign names and create variables.

We use a single equals sign to assign labels to variables. Let's see a few examples of how we can do this.

```
In [15]: # Let's create an object called "a" and assign it the number 5
a = 5
```

Now if I call **a** in my Python script, Python will treat it as the number 5.

```
In [16]: # Adding the objects
a+a
```

```
Out[16]: 10
```

What happens on reassignment? Will Python let us write it over?

```
In [17]: # Reassignment
a = 10
```

```
In [18]: # Check
a
```

```
Out[18]: 10
```


Mini Challenge - 1

It's your turn now!! store the word **hello** in my_string. print the my_string + name.

Mini Challenge - 2

It's your turn now!!! given the numbers stored in variables **a** and **b**. Can you write a simple code to compute the mean of these two numbers and assign it to a variable **mean**.

Practical-Tip

The names you use when creating these labels need to follow a few rules:

1. Names can not start with a number.
2. There can be no spaces in the name, use **_** instead.
3. Can't use any of these symbols : " , < > / \ () @ % * & ~ + -

Using variable names can be a very useful way to keep track of different variables in Python. For example:

From Sales to Data Science

Discover the story of Sagar Dawda who made a successful transition from Sales to Data Science. Making a successful switch to Data Science is a game of Decision and Determination. But it's a long road from Decision to Determination. To read more, click [here](#)

CHAPTER - 2 : Data Types & Data Structures

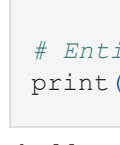
- Everything in Python is an "object", including integers/floats
- Most common and important types (classes)
 - "Single value": None, int, float, bool, str, complex
 - "Multiple values": list, tuple, set, dict
- Single/Multiple isn't a real distinction, this is for explanation
- There are many others, but these are most frequently used

Identifying Data Types

```
In [19]: a = 42
b = 32.30

print(type(a)) #gets type of a
print(type(b)) #gets type of b
```

```
<class 'int'>
<class 'float'>
```



Technical-Alert

Single Value Types

- int: Integers
- float: Floating point numbers
- bool: Boolean values (True, False)
- complex: Complex numbers
- str: String



Technical-Alert

Lists

Lists can be thought of the most general version of a **sequence** in Python. Unlike strings, they are mutable, meaning the elements inside a list can be changed!

In this section we will learn about:

- 1.) Creating Lists
- 2.) Indexing and Slicing Lists
- 3.) Basic List Methods
- 4.) Nesting Lists
- 5.) Introduction to List Comprehensions

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

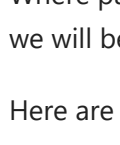
We just created a list of integers, but lists can actually hold different object types. For example:

```
In [21]: 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 [22]: len(my_list)
```

```
Out[22]: 4
```



Technical-Alert

Adding New Elements to a list

We use two special commands to add new elements to a list. Let's make a new list to remind ourselves of how this works:

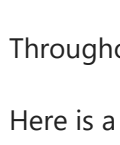
```
In [23]: my_list = ['one','two','three',4,5]
```

```
In [24]: # append a value to the end of the list
l = [1, 2, 3, 'a', 'b'], 'New York'
l.append(3.1)
print(l)
```

```
[1, 2, 3, 'a', 'b', 'New York', 3.1]
```

```
In [25]: # extend a list with another list.
l = [1, 2, 3]
l.extend([4, 5, 6])
print(l)
```

```
[1, 2, 3, 4, 5, 6]
```



Technical-Alert

Slicing

Slicing is used to access individual elements or a range of elements in a list.

Python supports "slicing" indexable sequences. The syntax for slicing lists is:

- **list_object[start:end:step]** or
- **list_object[start:end]**

start and end are indices (start inclusive, end exclusive). All slicing values are optional.

```
In [26]: list = list(range(10)) # create a list containing 10 numbers starting from 0
print(list)

print("Elements from index 4 to 7:", list[4:7])

print("Alternate elements, starting at index 0:", list[0:2]) # prints elements from index 0 till last index with
print("Every third element, starting at index 1:", list[1::3]) # prints elements from index 1 till last index with
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
elements from index 4 to 7: 4, 5, 6]
alternate elements, starting at index 0: [0, 2, 4, 6, 8]
every third element, starting at index 1: [1, 4, 7]
```

Other list operations

- **..append:** add element to end of list
- **..insert:** insert element at given index
- **..extend:** extend one list with another list



Technical-Alert

Dictionaries

Now we're going to switch gears and learn about **mappings** called **dictionaries** in Python. If you're familiar with other languages you can think of these Dictionaries as hash tables.

This section will serve as a brief introduction to dictionaries and consist of:

- 1.) Constructing a Dictionary
- 2.) Accessing objects from a dictionary
- 3.) Nesting Dictionaries
- 4.) Basic Dictionary Methods

A Python dictionary consists of a key and then an associated value. That value can be almost any Python object.

Constructing a Dictionary

Let's see how we can construct dictionaries to get a better understanding of how they work!

```
In [27]: # Make a dictionary with {} and : to signify a key and a value
my_dict = {'key1':'value1','key2':'value2'}
```

```
In [28]: # Call values by their key
my_dict['key2']
```

```
Out[28]: 'value2'
```

We can effect the values of a key as well. For instance:

```
In [29]: my_dict['key1']=123
my_dict
```

```
Out[29]: {'key1': 123, 'key2': 'value2'}
```

```
In [30]: # Subtract 123 from the value
my_dict['key1'] = my_dict['key1'] - 123
```

```
In [31]: #Check
my_dict['key1']
```

```
Out[31]: 0
```

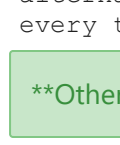
A quick note, Python has a built-in method of doing a self subtraction or addition (or multiplication or division). We could have also used += or -= for the above statement. For example:

```
In [32]: # Set the object equal to itself minus 123
my_dict['key1'] -= 123
my_dict['key1']
```

```
Out[32]: -123
```

Now it's your turn to get hands-on with Dictionary, create a empty dicts. Create a new key calle animal and assign a value 'Dog' to it..

```
In [33]: # Create a new dictionary
d = {}
# Create a new key through assignment
d['animal'] = 'Dog'
```



Technical-Alert

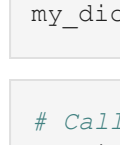
Set and Booleans

There are two other object types in Python that we should quickly cover. Sets and Booleans.

Sets

Sets are an unordered collection of **unique** elements. We can construct them by using the set() function. Let's go ahead and make a set to see how it works

Set Theory



```
In [34]: x = set()

# We add to sets with the add() method
x.add(1)

#Show
x
```

```
Out[34]: {1}
```

Note the curly brackets. This does not indicate a dictionary! Although you can draw analogies as a set being a dictionary with only keys.

We know that a set has only unique entries. So what happens when we try to add something that is already in a set?

```
In [35]: # Add a different element
x.add(2)
```

```
#Show
x
```

```
Out[35]: {1, 2}
```

```
In [36]: # Try to add the same element
x.add(1)
```

```
#Show
x
```

```
Out[36]: {1, 2}
```

Notice how it won't place another 1 there. That's because a set is only concerned with unique elements! We can cast a list with multiple repeat elements to a set to get the unique elements. For example:

```
In [37]: # Create a list with repeats
l = [1,1,2,2,3,4,5,6,1,1]

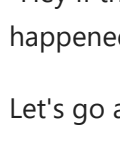
# Cast as set to get unique values
set(l)
```

```
Out[38]: {1, 2, 3, 4, 5, 6}
```

CHAPTER - 3 : Python Programming Constructs

We'll be talking about

- Looping
- Conditional Statements
- Comprehensions



Technical-Alert

Loops and Iterative Statements

If,elif,else Statements

if Statements in Python allows us to tell the computer to perform alternative actions based on a certain set of results.

Verbally, we can imagine we are telling the computer:

"Hey if this case happens, perform some action"

We can then expand the idea further with elif and else statements, which allow us to tell the computer:

"Hey if this case happens, perform some action. Else if another case happens, perform some other action. Else-- none of the above cases happened, perform this action"

Let's go ahead and look at the syntax format for if statements to get a better idea of this:

```
if case1:
    perform action1
elif case2:
    perform action2
else:
    perform action 3
```

```
In [39]: a = 5
b = 4

if a > b:
    # we are inside the if block
    print("a is greater than b")
elif b > a:
    # we are inside the elif block
    print("b is greater than a")
else:
    # we are inside the else block
    print("a and b are equal")

# Note: Python doesn't have a switch statement

a is greater than b
```



Indentation


```
In [41]: list1 = [4, 7, 13, 3, 11, 15]
list2 = []

for index, e in enumerate(list1):
    if e == 10:
        break
    if e < 10:
        continue
    list2.append((index, e*e))
else:
    print("out of loop without using break statement")

list2

out of loop without using break statement
Out[41]: [(2, 169), (3, 121), (5, 121), (6, 225)]
```

Technical-
Stuff

While loops

The **while** statement in Python is one of most general ways to perform iteration. A **while** statement will repeatedly execute a single statement or group of statements as long as the condition is true. The reason it is called a 'loop' is because the code statements are looped through over and over again until the condition is no longer met.

The general format of a while loop is:

```
while test:
    code statement
else:
    final code statements
```

Let's look at a few simple while loops in action.

```
In [43]: x = 0

while x < 10:
    print ('x is currently: ',x,end=' ') #end= ' ' to put print below statement on the same line after this stat
    print (' ' x is still less than 10, adding 1 to x')
    x+=1

x is currently: 0 x is still less than 10, adding 1 to x
x is currently: 1 x is still less than 10, adding 1 to x
x is currently: 2 x is still less than 10, adding 1 to x
x is currently: 3 x is still less than 10, adding 1 to x
x is currently: 4 x is still less than 10, adding 1 to x
x is currently: 5 x is still less than 10, adding 1 to x
x is currently: 6 x is still less than 10, adding 1 to x
x is currently: 7 x is still less than 10, adding 1 to x
x is currently: 8 x is still less than 10, adding 1 to x
x is currently: 9 x is still less than 10, adding 1 to x

Technical-
Stuff
```

Comprehensions

- Python provides syntactic sugar to write small loops to generate lists/sets/tuples/dicts in one line
- These are called comprehensions, and can greatly increase development speed and readability

Syntax:

sequence = [expression(element) for element in iterable if condition]

The brackets used for creating the comprehension define what type of object is created.

Use [] for lists, {} for generators, {} for sets and dicts

```
In [44]: list Comprehension

names = ["Ravi", "Pooja", "Vijay", "Kiran"]
hello = ["Hello " + name for name in names]
print(hello)

['Hello Ravi', 'Hello Pooja', 'Hello Vijay', 'Hello Kiran']

In [45]: numbers = [35, 32, 87, 99, 10, 54, 32]
even = [num for num in numbers if num % 2 == 0]
print(even)

odd_squares = [(num, num * num) for num in numbers if num % 2 == 1]
print(odd_squares)

[32, 10, 54, 32]
[(55, 3025), (87, 7569), (99, 9801)]

Technical-
Stuff
```

Exception Handling

try and except

The basic terminology and syntax used to handle errors in Python is the **try** and **except** statements. The code which can cause an exception to occur is put in the try block and the handling of the exception is implemented in the except block of code. The syntax form is:

```
try:
    You do your operations here...
...
except ExceptionI:
    If there is ExceptionI, then execute this block.
except ExceptionII:
    If there is ExceptionII, then execute this block.
...
else:
    If there is no exception then execute this block.
```

We can also just check for any exception with just using except. To get a better understanding of all this lets check out an example: We will look at some code that opens and writes a file:

```
In [46]: try:
        x = 1 / 0
    except ZeroDivisionError:
        print('divided by zero')
        print("executed when exception occurs")
    else:
        print("executed only when exception does not occur")
    finally:
        print("finally block, always executed")

divided by zero
executed when exception occurs
finally block, always executed

Concept-
Alert
```

Modules, Packages, and import

A module is a collection of functions and variables that have been bundled together in a single file. Module helps us:

- Used for code organization, packaging and reusability
- Module: A Python file
- Package: A folder with an `__init__.py` file
- Namespace is based on file's directory path

Module's are usually organised around a theme. Let's see how to use a module. To access our module we will import it using python's import statement. Math module provides access to the mathematical functions.

```
In [47]: # import the math module
import math

# use the log10 function in the math module
math.log10(123)

Out[47]: 2.089905111439398

Concept-
Alert
```

File I/O : Helps you read your files

- Python provides a **file** object to read text/binary files.
- This is similar to the **FileStream** object in other languages.
- Since a **file** is a resource, it must be closed after use. This can be done manually, or using a context manager (**with** statement)

```
In [48]: Create a file in the current directory

with open('myfile.txt', 'w') as f:
    f.write("This is my first file!\n")
    f.write("Second line!\n")
    f.write("Last line!\n")

# let's verify if it was really created.
# For that, let's find out which directory we're working from
import os
print(os.path.abspath(os.curdir))

C:\Users\Wishant\Downloads\Week-01-Git_4_Python_Intro\Week-01-Git_4_Python_Intro\notebooks

Read the newly created file

In [49]: # read the file we just created
with open('myfile.txt', 'r') as f:
    for line in f:
        print(line)

This is my first file!
Second line!
Last line!

Recap
```

In-session Recap Time

- Python Basics
 - Variables and Scoping
 - Modules, Packages and Imports
 - Data Types & Data Structures
- Python Programming Constructs
 - Data Types & Data Structures
 - Lists
 - Dictionaries
 - Sets & Booleans
 - Python Programming constructs
 - Loops and Conditional Statements
 - Exception Handling
 - File I/O

Thank You

Coming up next...

- Python Functions:** How to write modular functions to enable code reuse
- NumPy:** Learn the basis of most numeric computation in Python