Python is a modern, robust, high level programming language. It is very easy to pick up even if you are completely new to programming.

Python, similar to other languages like Matlab or R, is **interpreted** hence runs slowly compared to **compiled** languages like C++, Fortran or Java. However writing programs in Python is very quick. Python has a very large collection of libraries for everything from scientific computing to web services.

These lectures are using **ipython notebooks** which mix Python code with documentation. The python notebooks can be run on a webserver or stand-alone on a computer.

**1 Basic Syntax**

* Python has no **mandatory statement termination character**.
* **No spaces or tab** characters allowed at the **start of a statement**, except for indented blocks. (more on this later)
* Indentation plays a special role in Python, **blocks are specified by indentation**.
* The **#** character indicates that the rest of the line is a **comment**, **'''** represents multi-line comment.
* Values are **assigned** (in fact, objects are bound to names) with **=**.

1. var = 1

# var = 2.

''' The above statement is a comment,

This is a multi-line comment.

'''

print("value of var is %d." % var)

O/p :

value of var is 1.

2. var = 3 # no space at beginning

print(var) # indented code

File "<ipython-input-2-e30262a4614a>", line [2](https://localhost:8080/)

print(var) # indented code

^

IndentationError: unexpected indent

### 1.1 Variables and Types

Python is strongly typed (i.e. types are enforced), dynamically, implicitly typed (i.e. you don’t have to declare variables), case sensitive (i.e. var and VAR are two different variables) and object-oriented (i.e. everything is an object). The basic types build into Python include **float** (floating point numbers), **int** (integers), **str**(unicode character strings) and **bool** (boolean). The **type()** function returns the type of an object. Some examples of each:

3. var1 = 5

var2 = 15.2632423

var3 = "Hello World!"

var4 = True

print("var1 is of type %s." % type(var1))

print("var2 is of type %s." % type(var2))

print("var3 is of type %s." % type(var3))

print("var4 is of type %s." % type(var4))

o/p

var1 is of type <class 'int'>.

var2 is of type <class 'float'>.

var3 is of type <class 'str'>.

var4 is of type <class 'bool'>.

### 1.2 Help

Python has extensive help built in. You can execute **help()** for an overview or **help(x)** for any library, object or type x to get more information.

4. help(var1)

**Built-in Operators**

|  |  |  |
| --- | --- | --- |
| **Arithmetic** | **Logical** | **Bitwise** |
| |  |  | | --- | --- | | **Symbol** | **Task Performed** | | + | addition | | - | subtraction | | / | division | | // | floor division | | % | mod | | \* | multiplication | | \*\* | exponentiation | | |  |  | | --- | --- | | **Symbol** | **Task Performed** | | == | equals | | != | not equals | | < | less than | | > | greater than | | <= | less than or equal to | | >= | greater than or equal to | | |  |  | | --- | --- | | **Symbol** | **Task Performed** | | & | AND | | | | OR | | ^ | XOR | | ~ | NOT | | >> | right shift | | << | left shift | |

5. int1 = 5

int2 = 3

print("%d + %d equals %d." % (int1, int2, int1 + int2))

print("%d - %d equals %d." % (int1, int2, int1 - int2))

print("%d / %d equals %d." % (int1, int2, int1 / int2))

print("%d // %d equals %d." % (int1, int2, int1 // int2))

print("%d %% %d equals %d." % (int1, int2, int1 % int2))

print("%d \* %d equals %d." % (int1, int2, int1 \* int2))

print("%d \*\* %d equals %d." % (int1, int2, int1 \*\* int2))

5 + 3 equals 8.

5 - 3 equals 2.

5 / 3 equals 1.

5 // 3 equals 1.

5 % 3 equals 2.

5 \* 3 equals 15.

5 \*\* 3 equals 125.

6. int1 = 3

print("%d == 2 equals %r." %(int1, int1 == 2))

print("%d ! 2 equals %r." %(int1, int1 != 2))

print("%d < 2 equals %r." %(int1, int1 < 2))

print("1 < %d <= 10 equals %r." %(int1, 1 < int1 <= 10))

output

3 == 2 equals False.

3 ! 2 equals True.

3 < 2 equals False.

1 < 3 <= 10 equals True.

7. int1 = 2

int2 = 3

print("%d is %s in binary." %(int1, bin(int1)))

print("%d is %s in binary." %(int2, bin(int2)))

print("%d & %d is %s in binary." % (int1, int2, bin(int1 & int2)))

print("%d | %d is %s in binary." % (int1, int2, bin(int1 | int2)))

print("%d ^ %d is %s in binary." % (int1, int2, bin(int1 ^ int2)))

o/p

2 is 0b10 in binary.

3 is 0b11 in binary.

2 & 3 is 0b10 in binary.

2 | 3 is 0b11 in binary.

2 ^ 3 is 0b1 in binary.

## 2 Data Structure

### 2.1 String

Strings have to be enclosed in **'** single or **"** double quotation marks, and you can have quotation marks of one kind inside a string that uses the other kind. Multiline strings are enclosed in **'''** or **"""** triple quotes.

Strings are **immutable** i.e. is not possible to modify a string. All string modification operations create a new string and return it.

There are lots of **built-in methods** for formating and manipulating strings built into Python. Some of these are illustrated here.

1. String concatenation is the "addition" of two strings. (There is no subtraction.)
2. Multiplying a string by an integer simply repeats it.
3. Strings can be compared in lexicographical order with the usual comparisons.
4. Strings can be indexed with square brackets. Indexing starts from zero in Python and -1 represents the last index.

8. str1 = 'Hello'

str2 = 'World!'

str3 = "Hello World!"

str4 = '''Multi-line

string,

three lines to be exact.'''

print("'%s' + '%s' equals '%s'." % (str1, str2, str1 + str2))

print("'%s' \* 5 equals '%s'." %(str3, str3\*5))

o/p

'Hello' + 'World!' equals 'HelloWorld!'.

'Hello World!' \* 5 equals 'Hello World!Hello World!Hello World!Hello World!Hello World!'.

9. print("'%s' < '%s' equals %r." % (str1, str2, str1 < str2))

str5 = 'Python'

print("First character of '%s' is '%s'." % (str5, str5[0]))

print("Last character of '%s' is '%s'." % (str5, str5[-1]))

print("First three characters of '%s' is '%s'." % (str5, str5[0:3]))

o/p

'Hello' < 'World!' equals True.

First character of 'Python' is 'P'.

Last character of 'Python' is 'n'.

First three characters of 'Python' is 'Pyt'.

10. str1="hello World"

print("'%s' capitalized is '%s'." % (str1, str1.capitalize()))

print("'%s' in upper case is '%s'." % (str1, str1.upper()))

print("'%s' in lower case is '%s'." % (str1, str1.lower()))

o/p

'hello World' capitalized is 'Hello world'.

'hello World' in upper case is 'HELLO WORLD'.

'hello World' in lower case is 'hello world'.

11. str1=" hello world with trailing space"

print("'%s' stripped of space is \n'%s'.\n" % (str1, str1.strip())) # strip can also remove characters other than space.

str2 = str1.strip()

print("Replacing 'with' with 'without' in '%s' gives \n'%s'.\n" % (str2, str2.replace("with", "without")))

print("'%s' splitted by space is \n'%s'.\n" % (str2, str2.split())) # split can divide based on characters other than space.

o/p

' hello world with trailing space' stripped of space is

'hello world with trailing space'.

Replacing 'with' with 'without' in 'hello world with trailing space' gives

'hello world without trailing space'.

'hello world with trailing space' splitted by space is

'['hello', 'world', 'with', 'trailing', 'space']'.

12. str1='Python'

str2='C'+str1[1:]

print(str1,'-->',str2)

str3=str1.replace('P','C')

print(str1,'-->',str3)

str1[0] = 'C'

o/p

Python --> Cython

Python --> Cython

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

TypeError Traceback (most recent call last)

<ipython-input-13-0d9a91fc0a27> in <module>()

**7** print(str1,'-->',str3)

**8**

----> [9](https://localhost:8080/) str1[0] = 'C'

TypeError: 'str' object does not support item assignment

#### 2.1.1 Print Statement

The **print()** function prints all of its arguments as strings, separated by spaces and followed by a linebreak:

- print("Hello World")

- print("Hello",'World')

- print("Hello", <Variable Containing the String>)

To fill a string with values from variables, you use the % (modulo) operator and a tuple. Some common string formatters are:

* %s -> string
* %d -> Integer
* %f -> Float

str1 = "World"

print("Hello %s" % str1)

print("Hello", str1)

print("Actual Number = %d" %18)

print("Float of the number = %f" %18)

o/p

Hello World

Hello World

Actual Number = 18

Float of the number = 18.000000

### 2.2 Lists

Lists are the most commonly used data structure. Think of it as a **1D array**, where each element can be accessed by calling it's index value. Lists are **dynamic** in nature i.e. you can keep on adding data to a list. Lists need not be homogeneous, i.e. the same list can contain elements of different data types, even another list.

Lists are declared by **[]** or **list()**.

list1 = []

list2 = list()

list3 = ['apple', 'orange']

list4 = [1, 2, 3]

list5 = ['apple', 2, 3, ['orange', 3, 5]]

print("list5 contains %s and is of type %s." % (list5, type(list5)))

print("First element of list5 is %s of type %s." % (list5[0], type(list5[0])))

print("Last element of list5 is %s of type %s." % (list5[-1], type(list5[-1])))

print("First element of the above nested list is %s." % list5[-1][0])

o/p

list5 contains ['apple', 2, 3, ['orange', 3, 5]] and is of type <class 'list'>.

First element of list5 is apple of type <class 'str'>.

Last element of list5 is ['orange', 3, 5] of type <class 'list'>.

First element of the above nested list is orange.

Python Program to print the largest even and largest odd number in a list.

n=int(input("Enter the number of elements to be in the list:"))

b=[]

**for** i **in** range(0,n):

a=int(input("Element: "))

b.append(a)

c=[]

d=[]

**for** i **in** b:

**if**(i%2==0):

c.append(i)

**else**:

d.append(i)

c.sort()

d.sort()

count1=0

count2=0

**for** k **in** c:

count1=count1+1

**for** j **in** d:

count2=count2+1

**print**("Largest even number:",c[count1-1])

**print**("Largest odd number",d[count2-1])

#### 2.2.1 Slicing

Indexing is limited to accessing a single element, slicing on the other hand is accessing a (sub)sequence of data inside the list. It is written as **[a:b:c]** where a,b are the index values from the parent list and c is the step size.

list6 = [1,2,3,4,5,6,7,8,9,10]

print("The first to fourth elements of list6 are %s." % list6[0:4])

print("all but last element of list6 are %s." % list6[:-1])

print("every even index element of list6 are %s." % list6[::2])

o/p

The first to fourth elements of list6 are [1, 2, 3, 4].

all but last element of list6 are [1, 2, 3, 4, 5, 6, 7, 8, 9].

every even index element of list6 are [1, 3, 5, 7, 9].

#### 2.2.2 Built-in Functions

* To find the length of the list or the number of elements in a list, **len()** is used.
* **append()** is used to add a single element at the end of the list.
* Appending a list to a list would create a sublist. If a nested list is not what is desired then the **extend()** function can be used.
* **insert(x,y)** inserts but does not replace element. If you want to replace the element with another element you simply assign the value to that particular index.
* **count()** is used to count the number of a particular element that is present in the list.
* **index()** is used to find the index value of a particular element. Note that if there are multiple elements of the same value then the first index value of that element is returned.

There are a ton of other useful built-in functions like **max()**, **min()**, **pop()**, **sort()**, type **help(list)**.

list7 = [1, 2, 3, 4, 5, 6]

print("List contains", list7)

list7.append(7)

print("Appending 7 to the list gives", list7)

list7.extend([8, 9, 10])

print("Extending the list by several numbers gives", list7)

list7.insert(5, 6)

print("Inserting a 6 at index 5 gives", list7)

print("The element 6 occurs %s times." % list7.count(6))

print("The length of the list is", len(list7))

o/p

List contains [1, 2, 3, 4, 5, 6]

Appending 7 to the list gives [1, 2, 3, 4, 5, 6, 7]

Extending the list by several numbers gives [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Inserting a 6 at index 5 gives [1, 2, 3, 4, 5, 6, 6, 7, 8, 9, 10]

The element 6 occurs 2 times.

The length of the list is 11

### 2.3 Tuples

Tuples are similar to lists but are immutable i.e. the elements inside a tuple cannot be changed.

Tuples are declared by **()** or **tuple()** or by ending a sequence with a **,**.

tup1 = []

tup2 = tuple()

tup3 = ('apple', 'orange')

print(type(tup3), tup3)

tup4 = 1,

print(type(tup4), tup4)

o/p

<class 'tuple'> ('apple', 'orange')

<class 'tuple'> (1,)

### 2.4 Sets

Sets are mainly used to eliminate repeated numbers in a sequence/list.

Sets are declared as **set()** which will initialize a empty set. **set([sequence])** or **{sequence}** declares a set with elements.

Sets have built-in functions for standard set operations like **union()**, **intersection()**, **difference()**, **symmetric\_difference()**, **issubset()**, **isdisjoint()**, **issuperset()**, etc.

set1 = set()

print("type of set1 is", type(set1))

o/p

type of set1 is <class 'set'>

set2 = set([1, 2, 3, 4])

set3 = {3, 4, 5, 6, 7}

print("set2 contains", set2)

print("set3 contains", set2)

print("Union is", set2.union(set3)) # same as |

print("Intersection gives", set2.intersection(set3)) # same as &

print("Difference gives", set2.difference(set3)) # same as -

print("Symmetric difference gives", set2.symmetric\_difference(set3)) # same as ^

print("is set2 subset of set3?", set2.issubset(set3)) # same as <=

print("is set2 and set3 disjoint?", set2.isdisjoint(set3))

print("is set2 superset of set3?", set2.issuperset(set3)) # same as >=

print("is 4 in set2?", (5 in set2))\

o/p

set2 contains {1, 2, 3, 4}

set3 contains {1, 2, 3, 4}

Union is {1, 2, 3, 4, 5, 6, 7}

Intersection gives {3, 4}

Difference gives {1, 2}

Symmetric difference gives {1, 2, 5, 6, 7}

is set2 subset of set3? False

is set2 and set3 disjoint? False

is set2 superset of set3? False

is 4 in set2? False

### 2.5 Dictionaries

Dictionaries are mappings between keys and items. **Keys** in dictionaries are always **unique**.

Dictionaries are declared by **{}** or **dict()**.

dict1 = dict()

dict2 = {}

dict3 = {1: 'One', 2 : 'Two', 100 : 'Hundred'}

print("type of dict1 is", type(dict1))

o/p

type of dict1 is <class 'dict'>

Assign new (key, value) pairs to a dict using **dict[key] = value**. If key already exists the value is overwritten. **keys()** returns the list of all keys, **values()** return the list of all values and **items()** returns the list of all (key, value) pairs.

dict4 = {1: 'One', 'Two': 2, (1,2): 'tuple', '(1,2)': 'str'}

print("List of keys in dict4:", dict4.keys())

print("List of values in dict4:", dict4.values())

print("List of items in dict4:", dict4.items())

o/p

List of keys in dict4: dict\_keys([1, 'Two', (1, 2), '(1,2)'])

List of values in dict4: dict\_values(['One', 2, 'tuple', 'str'])

List of items in dict4: dict\_items([(1, 'One'), ('Two', 2), ((1, 2), 'tuple'), ('(1,2)', 'str')])

## 3 Control Flow Statements

The key thing to note about Python's control flow statements and program structure is that it uses **indentation** to mark blocks. Hence the amount of white space (space or tab characters) at the start of a line is very important.

### 3.1 If Else If

if some\_condition:

algorithm

elif some\_condition:

algorithm

else:

algorithm

if statements can be nested.

x = 10

y = 12

if x > y:

print("x>y")

elif x < y:

print("x<y")

else:

print("x=y")

o/p

x<y

### 3.2 For

for variable in something:

algorithm

When looping over integers the **range()** function is useful which generates a range of integers:

* range(n) = 0, 1, ..., n - 1
* range(m,n) = m, m + 1, ..., n - 1
* range(m,n,s) = m, m + s, m + 2s, ..., m + ((n - m - 1) // s) \* s

The **enumerate()** function gives **(index, element)** pairs while iterating over a sequence, instead of just the element.

list\_of\_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

total = 0

for each\_list in list\_of\_lists:

for num in each\_list:

total = total + num

print(total)

o/p

45

### 3.3 While

while some\_condition:

algorithm

i = 1

while i < 3:

print(i \*\* 2)

i = i+1

print('Bye')

o/p

1

4

Bye

### 3.4 Break

As the name says. It is used to break out of a loop when a condition becomes true when executing the loop.

for i in range(100):

print(i)

if i>=4:

break

o/p

0

1

2

3

4

### 3.5 Continue

This continues the rest of the loop. Sometimes when a condition is satisfied there are chances of the loop getting terminated. This can be avoided using continue statement.

for i in range(4):

if i == 2:

print("Ignored", i \*\* 2)

continue

print("Processed", i) # this statement is not reach if i > 4

o/p

Processed 0

Processed 1

Ignored 4

Processed 3

### 3.6 Catching exceptions

Some errors may not necessitate interrupting the program. A try block allows you to catch exceptions that happen anywhere during the exeuction of the try block and let you deal with it:

try:

code

except <Exception Type> as <variable name>:

# deal with error of this type

except:

# deal with any error

for i in [2,1.5,0.0,3]:

inv\_sum = 0

try:

inverse = 1.0/i

inv\_sum += inverse

except ZeroDivisionError as e: # no matter what exception

print(e)

print(inv\_sum)

o/p

float division by zero

0.3333333333333333

**4 Functions**

In programmming, functions are a mechansim to allow code to be re-used so that complex programs can be built up out of simpler parts. This is the basic syntax of a function:

def funcname(arg1, arg2,..., argN-i=<default\_value>, argN-i+1=<default\_value>,..., argN=<default\_value>):

statements

return <value>

Return values are optional (by default every function returns **None** if no return statement is executed)

x = range(10)

def func1(num\_list):

''' Function takes list of numbers as argument.

Return the highest, lowest, first and last element in the list.

'''

highest = max(num\_list)

lowest = min(num\_list)

first = num\_list[0]

last = num\_list[-1]

return highest, lowest, first, last

print("highest %d, lowest %d, first %d, last %d." % func1(x))

o/p

highest 9, lowest 0, first 0, last 9.

### 4.1 Scope of Variables

Whatever variable is declared inside a function is local variable and outside the function in global variable.

A global variable can be called from anywhere using the global keyword. Global values should be used sparingly as they make functions harder to re-use.

x = 1

y = 3

print("initiate x =", x)

print("initiate y =", y)

def func2():

global x

x = 2 # global variable

y = 4 # local variable

print("inside func x is", x)

print("inside func y is", y)

func2()

print("outside func x =", x)

print("outside func y =", y)

o/p

initiate x = 1

initiate y = 3

inside func x is 2

inside func y is 4

outside func x = 2

outside func y = 3

### 4.2 Lambda Functions

These are small functions which are not defined with any name and carry a single expression whose result is returned. Lambda functions come very handy when operating with lists. These functions are defined as:

lamba var1, var2, ..., varN: algorithm

The **map()** function takes as input a function and applies to a list/sequence

fahrenheit = [20, 25, 30]

celsius = map(lambda x: (float(9) / 5) \* x + 32, fahrenheit)

print(list(celsius))

o/p

[68.0, 77.0, 86.0]

### 4.3 Chaining Functions

In python functions can be chained i.e. you can call a function on the output of another function directly in a single line.

def square(num\_list):

for i, num in enumerate(num\_list):

num\_list[i] = (num \*\* 2)

return num\_list

def cumsum(num\_list):

cum\_sum = 0

for i, num in enumerate(num\_list):

cum\_sum += num

num\_list[i] = cum\_sum

return num\_list

def average(num\_list):

return sum(num\_list)/len(num\_list)

x = [1, 2, 3, 5, 8, 7, 6]

print(average(cumsum(square(x))))

o/p

71.71428571428571

### 4.4 List Comprehension

A very powerful concept in Python (that also applies to Tuples, sets and dictionaries as we will see below), is the ability to define lists using list comprehension (looping) expression.

x = [1, 2, 3, 5, 8, 7, 6]

print(average(cumsum([i \*\* 2 for i in x])))

o/p

71.71428571428571

Write python program to create a dictionary holding student details. The key is the register number. The values are name and marks of 3 subjects.

1. Compute the average marks of every student in the dictionary.

2. Display the student details whose names start with ‘A’.

3. Display those student details whose average is greater than 200.