

# Chapter 1: Python Basics

## 1 Introduction

### Introduction

Python is a dynamically typed programming language, meaning that we don't have to specify the type of function parameters or variables, the interpreter takes care of it.

## 2 Types

### Types

Like any programming language, Python offers two types of data types:

- **Primitive:** Stores simple, single values that are immutable.
- **Non-Primitive:** Stores collections of complex values that are usually mutable, with some exceptions.

### 2.1 Primitive Types

#### Primitive Types

- **int:** Integer numerical value.
- **float:** Decimal numerical value.
- **string:** Sequence of characters.
- **boolean:** Represents `True` or `False`.

### 2.2 Non-Primitive Types

#### Non-Primitive Types

- **list:** Stores an ordered collection of values.
- **tuple:** Similar to a list but we can't change their value directly only override the whole tuple with a new one.
- **set:** Stores a collection of unordered, unique values.
- **dictionary:** A mapping data structure where each element is a key-value pair.

## Primitive Example

```
1 age = 20
2 salary = 45532.21
3 name = "latex"
4 isTired = True
5
6 print(type(age))
7 print(type(salary))
8 print(type(name))
9 print(type(isTired))
```

```
rabah@UbuntuTex:~/Desktop/Documentations/University/3rd_
python/Chapters/Code/Basics/Types$ python3 prim.py
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

## Non Primitive Example

```
1 gradeList = [19, 16, 12, 11]
2 coordinatesTuple = (1, 2.5)
3 personMap = {"id": "1879", "data": "404 not found"}
4 uniqueNbSet = {1, 2, 3, 4}
5
6 print(type(gradeList))
7 print(type(coordinatesTuple))
8 print(type(personMap))
9 print(type(uniqueNbSet))
```

```
rabah@UbuntuTex:~/Desktop/Documentations/University/3rd_
python/Chapters/Code/Basics/Types$ python3 NonPrim.py
<class 'list'>
<class 'tuple'>
<class 'dict'>
<class 'set'>
```

## 3 Comments

### Comments

- Single Line Comment : they start with #
- Multi-Line Block Comment : wrapped between single or double quote : ''' , """

#### Example

```
1  ## this is a single comment
2  print("hello world")
3
4  '''
5  This is a
6  block comment
7  '''
8
9  """
10 This is another
11 block comment
12 """
```



A terminal window showing a user named 'rabah' on an 'UbuntuTex' system. The user is in the directory '/Desktop/Documentations/University/3rd/python/Chapters/Code/Basics/Comments'. They have executed the command 'python3 com.py', and the output is 'hello world'.

## 4 Input/Output

### IO

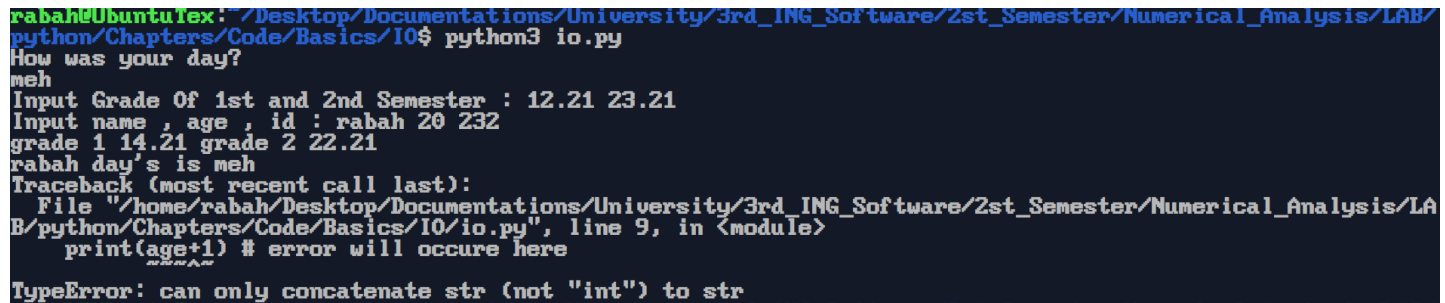
- print : we use the print function to display text , there are many string formatting we will see them in next section , the default formatting is text between qout and we seprate variable with comma , and it automatically add space
- input : has message inside ,by default input takes in string variable.we can type cast it , we can input many var at once with split() , we use map when inputing many var + typecasting

#### Syntax

```
1  var_1 = input ("message input")
2  var_2 , var_3 ... , var_n= input("message input").split()
3  casted_1 = Type(input("message input"))
4  casted_2, casted_3 , ... , casted_n = map(Type,input("message input").split())
5  print('text_1',var_1,'text_2',var_2,...,'text_n',var_n)
```

## Example

```
1 hru = input("How was your day?\n") #no need for casting
2
3 grade1,grade2 = map(float,input("Input Grade Of 1st and 2nd Semester : ").split()) # input 2 float we must type cast
4 name , age , id = input("Input name , age , id : ").split() # if we try arithmetic operation on age we will get error
5                                     # not casted to int
6
7 print('grade 1',grade1+2 , 'grade 2',grade2-1)
8 print(name,'day\'s is',hru)
9 print(age+1) # error will occure here
```



```
rabah@UbuntuTex: ~/Desktop/Documentations/University/3rd_ING_Software/2st_Semester/Numerical_Analysis/LAB/
python/Chapters/Code/Basics/I0$ python3 io.py
How was your day?
meh
Input Grade Of 1st and 2nd Semester : 12.21 23.21
Input name , age , id : rabah 20 232
grade 1 14.21 grade 2 22.21
rabah day's is meh
Traceback (most recent call last):
  File "/home/rabah/Desktop/Documentations/University/3rd_ING_Software/2st_Semester/Numerical_Analysis/LA
B/python/Chapters/Code/Basics/I0/io.py", line 9, in <module>
    print(age+1) # error will occure here
          ~~~^~
TypeError: can only concatenate str (not "int") to str
```

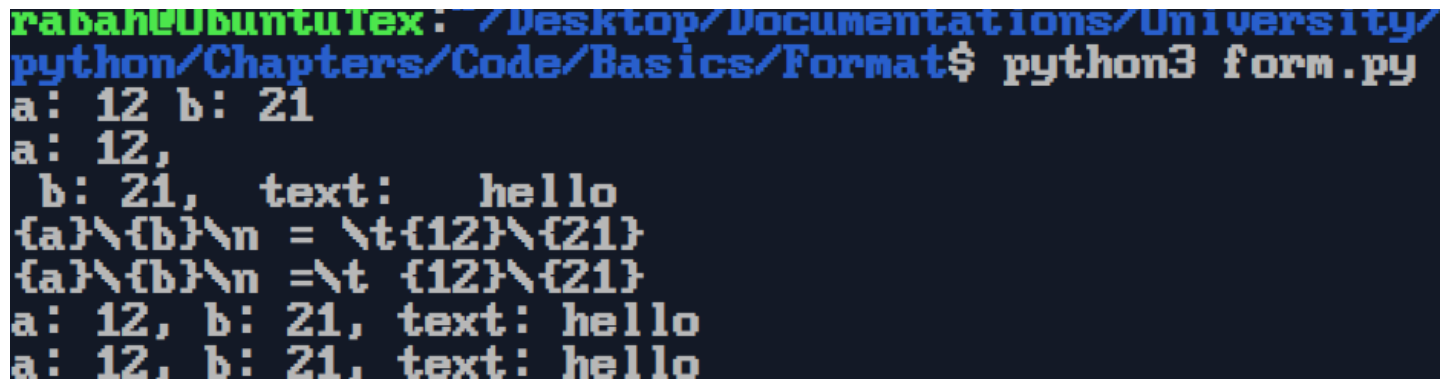
## 5 String Formatting

### Formatting

- Default : Uses `print()` with values separated by commas, automatically adding spaces.
- f-string : Uses `f"{var}"`, treats `\` as an escape character, and what's between `{}` is evaluated as a variable , Use `{{}}` for literal curly braces.
- raw-string : Uses `r"string"` to treat backslashes `\` as literal characters.
- f+raw-string : Uses `fr"string"`, supports `{var}` formatting of f-string while treating `\` as a literal character.
- % formatting : Uses `"format % value"`, similar to C-style formatting.
- .format : Uses `"{} {}".format(val1, val2)` to insert values into placeholders. Placeholders can be empty, indexed, or labeled.

## Example

```
1 a = 12
2 b = 21
3 text = "hello"
4
5 print("a:", a, "b:", b)                # default formatting
6
7 print(f"a: {a},\n b: {b},\t text:\t {text}") # f-string formatting
8
9 print(r"{a}\{b}\n = \t{12}\{21}")      # raw-string formatting
10
11 print(fr"{a}}\{b}}\n =\t {{a}}}\{b}}") # f+raw-string formatting
12
13 print("a: %d, b: %d, text: %s" % (a, b, text)) # %-formatting
14
15 print("a: {}, b: {}, text: {}".format(a, b, text)) # .format()
```



```
rabah@UbuntuTex: ~/Desktop/Documentations/University/
python/Chapters/Code/Basics/Format$ python3 form.py
a: 12 b: 21
a: 12,
  b: 21,  text:  hello
{a}\{b}\n = \t{12}\{21}
{a}\{b}\n =\t {12}\{21}
a: 12, b: 21, text: hello
a: 12, b: 21, text: hello
```

## 6 Importing Modules

### Import

A module in Python is simply a file containing Python code with classes , functions and variables that we can include and use in our own programs. There are two main ways to import modules:

- **Importing the entire module:** This loads all the module's functions, classes, and variables , to access them we need to write the modulename followed by a point
- **Importing specific parts of a module:** Instead of loading everything, we can import only specific functions, classes, or variables, which we call directly by their name.

## Note

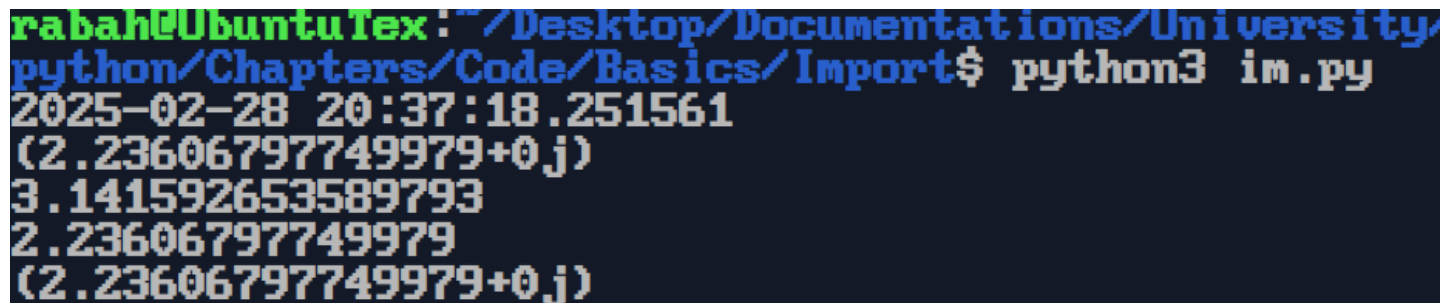
- We can rename imported modules or specific parts of a module using the `as` keyword.
- If we import multiple modules or module parts with the same name, Python will override previous imports with the latest one.
- One module can have submodules that can be accessed using `. : module.submodule`

## Syntax

```
1  # Note as are optional
2
3  import moduleName as moduleAlias  # import the whole module
4  from moduleName import Part_1 as PartAlias_1 , ... , Part_n as PartAlias_n  #import parts of the module
```

## Example

```
1  import datetime as dt          #import the whole datetime module as dt
2  import cmath                   #import the whole module cmath
3  import math                    #import the whole module math
4  from math import sqrt as sq , pi  #import sqrt function as sq and pi variable from math module
5  from cmath import sqrt as sq     #import sqrt function as sq from cmath module
6
7  print(dt.datetime.now())        #call the datetime.now() function of dt
8  print(cmath.sqrt(5))           #call the sqrt() function of camth
9  print(pi)                      #call the pi variable of math
10 print(math.sqrt(5))            #call the sqrt() function of math
11 print(sq(5))                  #call the sq() function of cmath (it was overridden with cmath)
```



```
rabah@UbuntuTex:~/Desktop/Documentations/University/python/Chapters/Code/Basics/Import$ python3 im.py
2025-02-28 20:37:18.251561
(2.23606797749979+0j)
3.141592653589793
2.23606797749979
(2.23606797749979+0j)
```

## 7 Operators

### Operators

- **Arithmetic Operators:**

- + : Addition
- - : Subtraction
- \* : Multiplication
- / : Float division
- // : Integer division
- % : Modulo
- \*\* : Exponentiation (power)
- = : Assignment

- **Comparison Operators:**

- == : Equal to
- != : Not equal to
- > : Greater than
- < : Less than
- >= : Greater than or equal to
- <= : Less than or equal to

- **Logical Operators:**

- and : Logical AND
- or : Logical OR
- not : Logical NOT

## 8 Control Structures

### Control Structures

Python does not use curly braces (`{}`) to define control structures, classes, or functions. Instead, it relies on indentation to determine code blocks 4 spaces. In this section, we will explore all control structures with syntax and examples.

#### 8.1 If Statement

##### Syntax

```
1 if condition:
2     # instructions indented with 2 spaces
3 elif condition:
4     # instructions indented with 2 spaces
5 else:
6     # instructions indented with 2 spaces
```

##### Example

```
1 a = 0;
2
3 if a > 0:
4     print(a, "is strictly positive")
5 elif a < 0:
6     print(a, "is strictly negative")
7 else:
8     print(a, "is null")
```

```
python/Chapters/Code/Basics/Control$ python3 if.py
0 is null
```



## 8.2 Match Statement(Switch Case)

### Syntax

```
1 match var:
2     case value_1:
3         # instructions indented with 8 spaces
4
5     .....
6
7     case value_n:
8         # instructions indented with 8 spaces
9
10    case default: # default if var isn't in (value_1,...,value_n)
11        # instructions indented with 8 spaces
```

### Example

```
1 a = 20
2
3 match a:
4     case -21:
5         print("negative")
6     case 2|10|21:
7         print("positive")
8     case int() :
9         print("integer")
10    case default :
11        print("idk")
```

```
python/Chapters/Code/Basics/Control$ python3 switch.py
integer
```

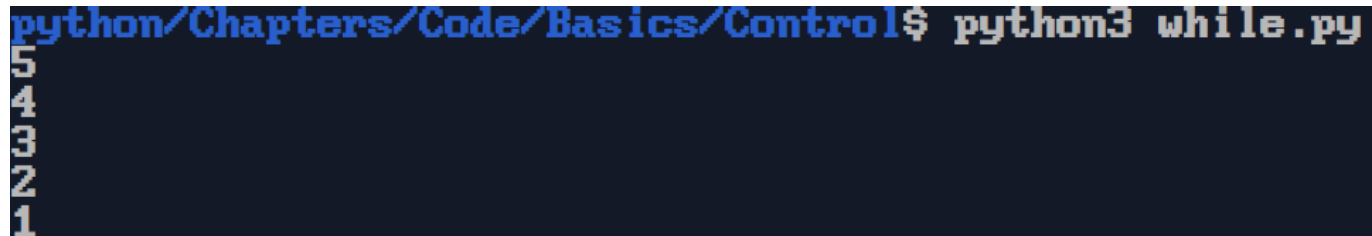
## 8.3 While Loop

### Syntax

```
1 while condition:
2     #instructions indented with 4 spaces
```

### Example

```
1 cpt = 5
2
3 while cpt > 0:
4     print(cpt)
5     cpt = cpt - 1;
```



A terminal window with a dark background and light blue text. The prompt is `python/Chapters/Code/Basics/Control$` followed by the command `python3 while.py`. The output of the script is the numbers 5, 4, 3, 2, and 1, each on a new line.

## 8.4 For Loop

### Syntax

```
1 for value in iterable: # Iterates over each item in an iterable (e.g., list, tuple, set, string, generator)
2     # Instructions indented with 4 spaces
3
4 for index in range(n): # Loops from 0 to (n-1), incrementing by 1
5     # Instructions indented with 4 spaces
6
7 for index in range(a, n): # Loops from a to (n-1), incrementing by 1
8     # Instructions indented with 4 spaces
9
10 for index in range(a, n, b): # Loops from a to (n-1), incrementing by b (can be negative for reverse loops)
11     # Instructions indented with 4 spaces
```

## Example

```
1 for value in "latex": # ['l','a','t','e','x']
2     print(value)
3
4 strList = ["hey","im","alright"]
5
6 print("\n")
7
8 for value in strList: # ["hey","im","alright"]
9     print(value)
10
11 print("\n")
12
13 n = len(strList)
14
15 for i in range(n): # ["hey","im","alright"]
16     print(strList[i])
17
18 print("\n")
19
20 for i in range(1,n): # ["im","alright"]
21     print(strList[i])
22
23 print("\n")
24
25 for i in range(0,n,2): # ["hey","alright"]
26     print(strList[i])
```

```
rabah@Ubuntu1ex: /Desktop/Documentations/University/3rd_1
python/Chapters/Code/Basics/Control$ python3 for.py
l
a
t
e
x

hey
im
alright

hey
im
alright

im
alright

hey
alright
```

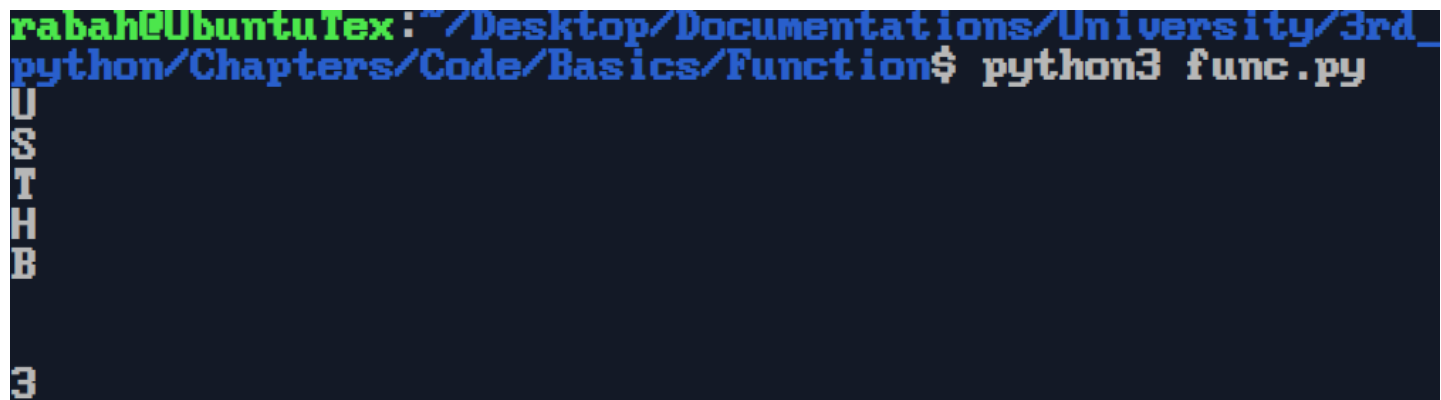
## 9 Function

### Syntax

```
1 def function_name():
2     # Instructions indented with 4 spaces
3     return value
4
5 def procedure_name():
6     # Instructions indented with 4 spaces
```

### Example

```
1 def sum(a,b):
2     return a+b
3
4 def print_char(string):
5     for char in string:
6         print(char)
7
8     print("\n")
9
10 print_char("USTHB")
11 s = sum(1,2)
12 print(s)
```



A terminal window screenshot showing the execution of a Python script. The prompt is 'rabah@UbuntuTex:~/Desktop/Documentations/University/3rd\_python/Chapters/Code/Basics/Function\$'. The command 'python3 func.py' has been executed. The output is displayed on the next line: 'USTHB' followed by a blank line and the number '3'.

## 10 NamedTuple

### NamedTuple

NamedTuples are similar to structures in C. The key difference is that they have the same properties as tuples, meaning their elements are immutable, so we cannot modify them directly only override them with new ones.

To create a NamedTuples, we need to import the `namedtuple` function from the `collections` module. It takes the name of the tuple and its attributes as parameters.

### Syntax

```
1 from collections import namedtuple
2
3 # Defining a NamedTuple with its name and attributes
4 StructName = namedtuple("StructName", ["attributeName_1", ..., "attributeName_n"])
5
6 # Create an instance of the NamedTuple
7 struct_var = StructName(value_1, ..., value_n)
8
9 # Accessing attributes with .
10 print(struct_var.attributeName_n)
```

### Example

```
1 from collections import namedtuple
2
3 intervalle = namedtuple("intervalle", ["a", "b"])
4
5 intervalle_var = intervalle(1,2)
6
7 print("[ a , b ] = [", intervalle_var.a, ",", intervalle_var.b, "]")
```



```
python/Chapters/Code/Basics/Struct$ python3 st.py
[ a , b ] = [ 1 , 2 ]
```

# Chapter 2: NumPy

## 1 Introduction

### Introduction

NumPy(Numerical Python) is a fundamental library for Python numerical computing. It provides efficient multi-dimensional array objects and various mathematical functions for handling large datasets.

To use NumPy, we need to import the `numpy` module. By convention, it is commonly aliased as `np` to simplify usage and improve readability.

## 2 Array

### Array

NumPy arrays are more efficient and faster than the collections we have seen so far. They allow operations to be performed on the entire array at once, eliminating the need for looping through elements individually. We can create NumPy arrays using built-in functions or convert existing lists into arrays using the `array` function.

### 2.1 Converting List To Array

#### Converting

To convert a list into an NumPy array we use the `array()` function.

#### Syntax

```
1 import numpy as np
2
3 npArray = np.array(List)
```

#### Example

```
1 import numpy as np
2
3 List = [1,2,3]
4
5 npArray_1 = np.array(List)
6 npArray_2 = np.array([4,2,-2])
```

## 2.2 Generating Arrays

### Generating

- `arange(start=0, stop, step=1, dtype=None)` : Creates an array of values with a specified step size.
  - **start** (default = 0) is optional the beginning of the sequence.
  - **stop** is the end value (not included in the array).
  - **step** (default = 1) determines the increment.
  - **dtype** is optional and defines the data type. If not specified, NumPy infers it automatically.
- `linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)` : Generates an array of num element evenly spaced values.
  - **start** is the first value in the sequence.
  - **stop** is the last value (included by default if **endpoint=True**).
  - **num** (default = 50) specifies the total number of values.
  - **endpoint** is optional (default = True) determines whether **stop** is included.
  - **retstep** is optional (default = False) returns the step size if set to True.
  - **dtype** is optional and specifies the data type.
  - **axis** is optional (default = 0) is used in multi-dimensional arrays.

### Arange()

#### Syntax

```
1 import numpy as np
2
3 Array_1 = np.arange(n)           # generate array from 0 to (n-1) with step = 1
4 Array_2 = np.arange(a, n)        # generate array from a to (n-1) with step = 1
5 Array_3 = np.arange(a, n, b)     # generate array from a to less than n with step = b
6 Array_4 = np.arange(stop=n, step=b) # generate array from 0 to less than n with step = b
```

### Example

```
1 import numpy as np
2
3 array_1 = np.arange(5)
4 array_2 = np.arange(stop=5, step=2)
5 array_3 = np.arange(1, 5, 2)
6 array_4 = np.arange(1, 5)
7
8 print(array_1)
9 print(array_2)
10 print(array_3)
11 print(array_4)
```

```
rabah@UbuntuTex:~/Desktop/Documentations/University/3rd_1
python/Chapters/Code/NP/Array$ python3 arr.py
[0 1 2 3 4]
[0 2 4]
[1 3]
[1 2 3 4]
```

## LinSpace()

### Syntax

```
1 import numpy as np
2
3 Array_1 = np.linspace(a, n)           # generate array from a to n with 50 elements (default)
4 Array_2 = np.linspace(a, n, b)       # generate array from a to n with b elements
5 Array_3 = np.linspace(a, n, b, False) # generate array from a to n (endpoint excluded) with b elements
6 Array_4, step = np.linspace(a, n, b, False, True) # generate array from a to n (endpoint excluded) with b elements
7                                           # also returns step size
```

### Example

```
1 import numpy as np
2
3 array_1 = np.linspace(1,2,5)
4 array_2 = np.linspace(1,2, 5, False)
5 array_3, step = np.linspace(1, 2, 5, False, True)
6 array_4 = np.linspace(1,2)
7
8 print(array_1)
9 print(array_2)
10 print(step, array_3)
11 print(array_4)
```

```
rabah@UbuntuTex:~/Desktop/Documentations/University/3rd_ING_Software/2st_5
python/Chapters/Code/NP/Array$ python3 lin.py
[1.  1.25 1.5  1.75 2.  ]
[1.  1.2 1.4 1.6 1.8]
0.2 [1.  1.2 1.4 1.6 1.8]
[1.  1.02040816 1.04081633 1.06122449 1.08163265 1.10204082
 1.12244898 1.14285714 1.16326531 1.18367347 1.20408163 1.2244898
 1.24489796 1.26530612 1.28571429 1.30612245 1.32653061 1.34693878
 1.36734694 1.3877551  1.40816327 1.42857143 1.44897959 1.46938776
 1.48979592 1.51020408 1.53061224 1.55102041 1.57142857 1.59183673
 1.6122449  1.63265306 1.65306122 1.67346939 1.69387755 1.71428571
 1.73469388 1.75510204 1.7755102  1.79591837 1.81632653 1.83673469
 1.85714286 1.87755102 1.89795918 1.91836735 1.93877551 1.95918367
 1.97959184 2.  ]
```



## 2.3 Array Slicing

### Slicing

```
array[start=0 : stop=len(array) : step=1]
```

- **start** (default = 0): The index where slicing begins. If negative, it is interpreted as `len(array) + start`.
- **stop** (default = `len(array)`): The index where slicing ends (exclusive). If negative, it is interpreted as `len(array) + stop`.
- **step** (default = 1): Determines the stride between elements. Can be negative for reverse slicing.

### Example

```
1 import numpy as np
2 arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
3
4 print(arr[:])
5 print(arr[2:7])
6 print(arr[1:7:2])
7 print(arr[:3])
8 print(arr[5:])
9 print(arr[:6])
10 print(arr[:-2])
11 print(arr[-3:])
```

```
rabah@UbuntuTex:~/Desktop/Documentations/University/
python/Chapters/Code/NP/Array$ python3 slice.py
[0 1 2 3 4 5 6 7 8 9]
[2 3 4 5 6]
[1 3 5]
[0 3 6 9]
[5 6 7 8 9]
[0 1 2 3 4 5]
[0 1 2 3 4 5 6 7]
[7 8 9]
```

### 3 Mathematical Function

#### Function

- `sin(x)` :  $\sin(x)$
- `cos(x)` :  $\cos(x)$
- `tan(x)` :  $\tan(x)$
- `exp(x)` :  $e^x$
- `log(x)` :  $\ln(x)$
- `log10(x)` :  $\log_{10}(x)$
- `log2(x)` :  $\log_2(x)$
- `log(x) / log(b)` :  $\log_b(x)$

# Chapter 3: Matplotlib

## 1 Introduction

### Introduction

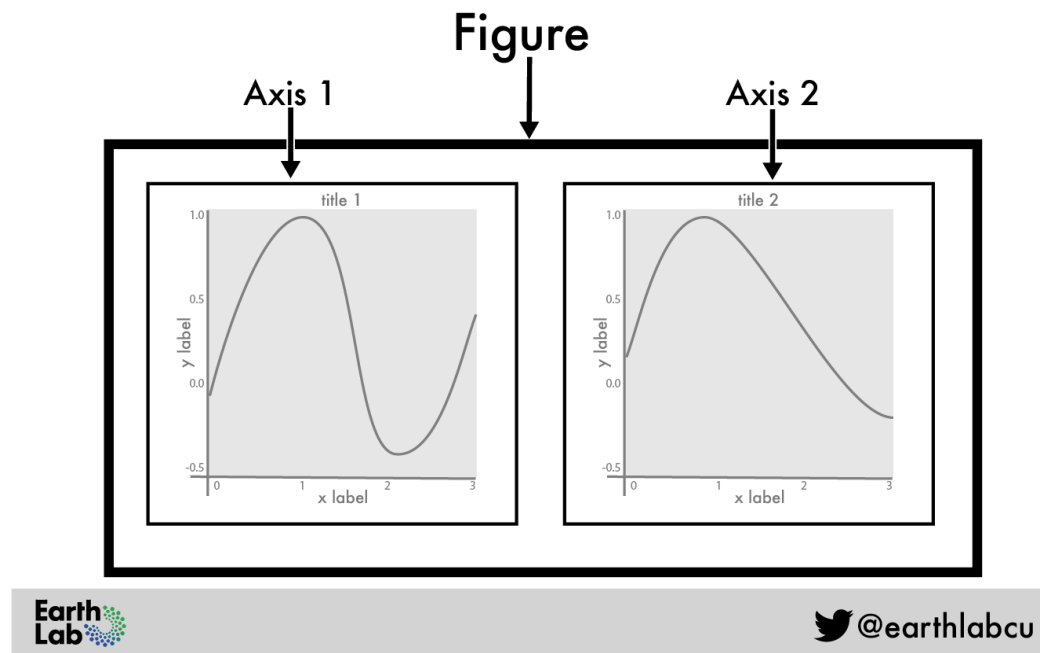
Matplotlib is a powerful Python library for data visualization. It allows users to create both static and interactive plots with ease. To use Matplotlib, we import the `pyplot` submodule, which provides a simple interface for plotting. By convention, it is commonly aliased as `plt` to enhance readability and simplify usage.

## 2 Figure & Axis

### Difference

A **Figure** is the top-level container that holds everything, similar to a window or a canvas. An **Axis** is a plotting area inside a Figure where data is drawn.

A single Figure can contain multiple Axes (subplots), allowing multiple plots within the same window.



### 3 Creating a Figure

#### Figure Creation

- A figure is created using the `figure()` function from the `pyplot` submodule.
- By default, Matplotlib starts with an implicit figure.
- Each additional call to `figure()` creates a new figure, so the total number of figures is :

$$1 + (\text{number of calls to figure()})$$

### 4 Drawing A Plot

#### Plot

`plot()` is a function from the `pyplot` submodule used to draw a graph on the axis of a figure.

```
plot(x, y, linestyle='-', linewidth=1.5, marker=None, markersize=6.0, color=auto, mfc=color, mec=color, label=None, alpha=1)
```

- **x**: A NumPy array representing the x-coordinates of the data points.
- **y**: A NumPy array representing the y-coordinates of the data points.
- **linestyle**: Optional parameter. Default is `'-'`. Specifies the style of the connecting line.
- **linewidth**: Optional parameter. Default is `1.5`. Controls the thickness of the line.
- **marker**: Optional parameter. Default is `None`. Defines the shape of markers placed at data points.
- **markersize**: Optional parameter. Default is `6.0`. Sets the size of the markers.
- **color**: Optional parameter. Default is `auto`. If not set, Matplotlib assigns a color automatically. Accepts color names, hex codes, and RGB(A) tuples.
- **mfc** (Marker Face Color): Optional parameter. Default is the same as `color`. Defines the fill color of the marker.
- **mec** (Marker Edge Color): Optional parameter. Default is the same as `color`. Defines the outline color of the marker.
- **label**: Optional parameter. Default is `None`. Specifies the legend label for the plot. Supports raw strings and LaTeX expressions using `$ $`.
- **alpha**: Optional parameter. Default is `1`. Controls the transparency of both lines and markers (`1` = fully opaque, `0` = fully transparent).

Marker value	Description	Appearance
'.'	Point	•
','	Pixel (Small Square)	▪
'o'	Circle	○
'v'	Triangle down	▼
'^'	Triangle up	▲
's'	Square	■
'*'	Star	★
'+'	Plus	+
'x'	Cross	×
'D'	Diamond	◆
none or ''	no marker (Default value)	nothing

linestyle value	Description	Appearance
'_'	Solid line (Default value)	_____
'--'	Dashed line	- - - - -
'-.'	Dash-dot line	- . - . - .
'...'	Dotted line	.....
none or ''	no line	nothing

## 5 Drawing a Single Point

### Single Point

We can use either the `plot()` or `scatter()` function:

- `plot(x, y, color=auto, linestyle='', marker, mfc=color, mec=color, alpha=1)`
- `scatter(x, y, color=auto, marker, c=color, edgecolors=color, alpha=1)` Here, `c` is equivalent to `mfc`, and `edgecolors` is equivalent to `mec`.

## 6 Graph Customization & Display

### Customization

- `legend()`: Displays the labels of plotted elements (e.g., lines, scatter points) as a legend for the axis.
- `grid(visible=True)`: Toggles the grid on the axis. By default, the grid is off, but calling the function without arguments is equivalent to setting it to `True`.
- `title(label)`: Sets the title for the current axis.
- `suptitle(label)`: Sets the title for the current figure.
- `ylabel(label)`: Labels the y-axis.
- `xlabel(label)`: Labels the x-axis.
- `show()`: Renders and displays all created figures along with their content.
- `plt.savefig(fname)`: saves the figure in the given path `fname`, if `fname` doesn't have a file extension it will save it as `png`

## 7 Subplot

### Subplot

The `subplot()` function allows us to create multiple axes within the same figure by defining a grid layout.

`subplot(nrows, ncols, index)`

- **nrows**: Number of rows in the subplot grid.
- **ncols**: Number of columns in the subplot grid.
- **index**: The position of the subplot, starting from 1 (left to right, top to bottom).

Each call to `subplot()` activates a different subplot within the figure.

## Example

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x1 = np.linspace(-2,2,100)
5 x2 = np.linspace(0.5,5,100)
6 x3 = np.linspace(-np.pi,np.pi,100)
7
8 y1 = np.exp(x1)
9 y2 = np.log(x2)
10 y3 = np.cos(x3)
11
12 plt.subplot(1,2,1)
13 plt.plot(x1,y1,label=r"$f_1(x) = e^x$",color = "red",linestyle='--',linewidth=1.75)
14 plt.title(r"$f_1(x)$")
15 plt.xlabel(r"$x_1$")
16 plt.ylabel(r"$y_1$")
17 plt.plot(np.array([0,1]),np.array([1,np.exp(1)]),linestyle=' ',color="lightblue",label="Points",marker='o')
18 plt.grid()
19 plt.legend()
20
21 plt.subplot(1,2,2)
22 plt.plot(x2,y2,label=r"$f_2(x) = \ln{(x)}$",color = "blue")
23 plt.title(r"$f_2(x)$")
24 plt.scatter(1,0,color="lightblue",label=r"$\alpha_1$",marker='*')
25 plt.scatter(2,np.log(2),color="lightgreen",label=r"$\alpha_2$",marker='D')
26 plt.legend()
27
28 plt.suptitle(r"figure$_1$")
29 plt.savefig("fig1.pdf")
30
31
32 plt.figure()
33
34 plt.plot(x3,y3,label=r"$f_3(x) = \cos{(x)}$",color = "green",linewidth=1.25)
35 plt.title(r"$f_3(x)$")
36 plt.grid(True)
37
38 plt.suptitle(r"figure$_2$")
39 plt.savefig("fig2.pdf")
40
41 plt.show()
```

fig1

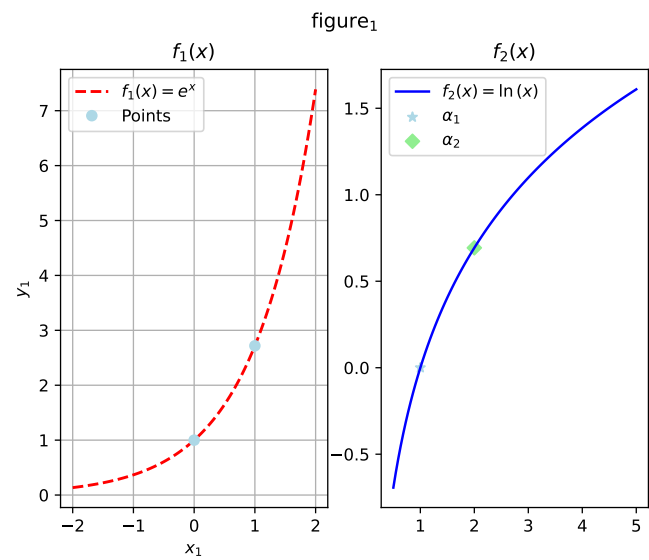
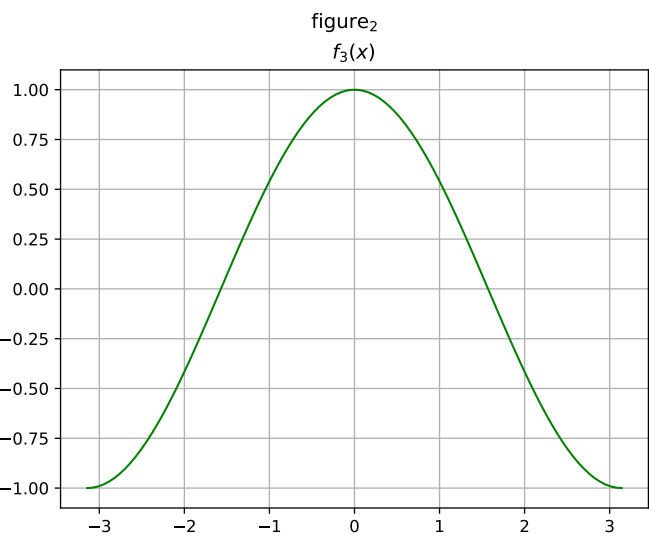


fig 2





## Note

We can remove the  $x, y$  axes of an axis from showing by using the `plt.axis('off')` function.

## 8 Drawing Tables

### Table

To draw a table on an axis, we use the `table()` function, which returns a `Table` object.

```
the_table = table(cellText, colLabels=None, rowLabels=None, cellLoc='right', fontsize=auto, colWidths=None, loc='bottom')
```

- **cellText**: A 2D list of strings that holds the content of each cell.
- **colLabels** (optional, default: `None`): A list of strings for the column headers.
- **rowLabels** (optional, default: `None`): A list of strings for the row headers.
- **cellLoc** (optional, default: `'right'`): Alignment of cell content. Possible values: `'right'`, `'left'`, `'center'`.
- **fontsize** (optional, default: `auto`): The font size of the table text.
- **colWidths** (optional): A list of floats representing the width of each column.
- **loc** (optional, default: `'bottom'`): The position of the table relative to the axes.

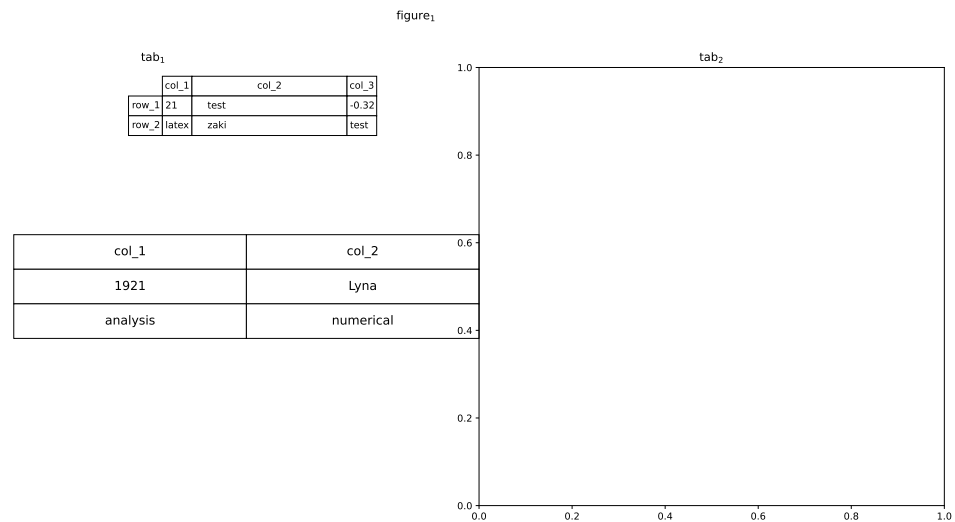
Some Function of `Table` object

- **the\_table.auto\_set\_column\_width(col=None)** : Adjusts the width of selected columns to fit their content.
  - **col** (optional, default: `None`): A list of integers representing the column indices to adjust.
- **the\_table.scale(xscale, yscale)** : Scales the table by adjusting cell padding.
  - **xscale** : Horizontal scaling factor.
  - **yscale** : Vertical scaling factor.
- **the\_table.auto\_set\_font\_size(scale=True)** : Shrink font size until text fit in cell note that scaling padding with scale or calling the auto set columns width might override the behaviour of this function since text will always fit, and this behaviour is on by default.
  - **scale** (optional , default = `True`) : boolean either `True` or `False`.
- **the\_table.set\_fontsize(size)** : Manually sets the font size of the table text.
  - **size** : float value.

## Example

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 plt.subplot(1,2,1)
5 cellText_1 = [["21","test","-0.32"] , ["latex" , "zaki" , "test"]]
6 rowLabels_1 = ["row_1","row_2"]
7 colLabels_1 = ["col_1","col_2","col_3"]
8 table_1 = plt.table(fontsize=12,cellText=cellText_1,rowLabels=rowLabels_1,colLabels=colLabels_1,cellLoc = 'left',loc=
9 'best')
10 table_1.auto_set_column_width ([0,2])
11 plt.axis('off')
12 plt.title(r"tab$_1$")
13
14 plt.subplot(1,2,2)
15 cellText_2 = [["1921","Lyna"] , ["analysis" , "numerical"]]
16 colLabels_2 = ["col_1","col_2"]
17 table_2 = plt.table(cellText=cellText_2,colLabels=colLabels_2,cellLoc = 'center',loc='left')
18 table_2.scale(xscale = 1 , yscale = 1.75)
19 table_2.set_fontsize(40)
20 table_2.auto_set_font_size()
21
22 plt.title(r"tab$_2$")
23
24 plt.suptitle(r"figure$_1$")
25
26 plt.show()
```

fig1



# Chapter 4: Dichotomy Method

## Implementation

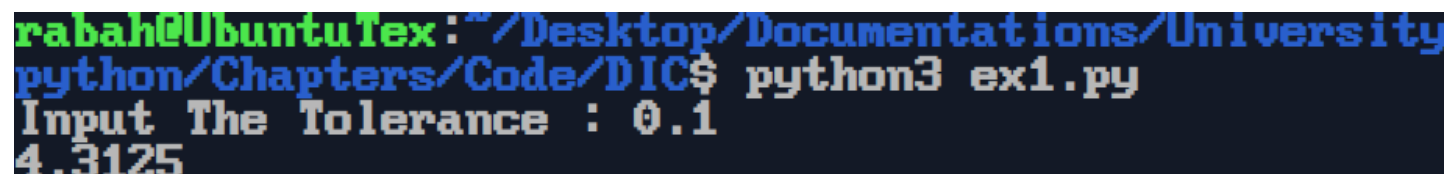
- `function(x)`: Returns the value of the function  $f(x)$  at a given point  $x$ .
- `ErrorEstimation(a, b)`: Estimates the error for the interval  $[a_n, b_n]$ , where  $n \geq 0$  :  $\frac{b_n - a_n}{2}$ .
- `eps`: Represents the error tolerance  $\epsilon$ .
- `max_iter`: Specifies the maximum number of iterations for the while loop.
- `dichotomy(eps, a, b, function, max_iter=100)`: Computes the root of the function  $f(x)$  using the dichotomy method over the interval  $[a, b]$ , with a given tolerance  $\epsilon$ , and max number of iteration (100 by default).

## Code

```
1 def function(x):
2     return ## function
3
4 def ErrorEstimation(a,b):
5     return (b-a)/2
6
7
8 def dichotomy(eps,a,b,function,max_iter=100):
9
10    n = 1
11
12    while ErrorEstimation(a,b) > eps and n <=max_iter :
13
14        x = (a+b)/2
15
16        if function(x) * function(a) < 0:
17            b = x
18        elif function(x) * function(b) < 0:
19            a = x
20        else:
21            return x
22
23        n = n+1
24
25    return (a+b)/2
26
27
28
29 eps = value
30 a = value_1
31 b = value_2
32 print(dichotomy(eps,a,b,function))
```

## Example

```
1 def function(x):
2     return x**3 - 80
3
4 def ErrorEstimation(a,b):
5     return (b-a)/2
6
7
8 def dichotomy(eps,a,b,function,max_iter=100):
9
10     n = 1
11
12     while ErrorEstimation(a,b) > eps and n <=max_iter :
13
14         x = (a+b)/2
15
16         if function(x) * function(a) < 0:
17             b = x
18         elif function(x) * function(b) < 0:
19             a = x
20         else:
21             return x
22
23         n = n+1
24
25     return (a+b)/2
26
27
28 eps = float(input("Input The Tolerance : "))
29 a = 4
30 b = 5
31 print(dichotomy(eps,a,b,function))
```



A terminal window screenshot showing the execution of the Python script. The prompt is 'rabah@UbuntuTex:~/Desktop/Documentations/University/python/Chapters/Code/DIC\$'. The command 'python3 ex1.py' is entered. The program prompts 'Input The Tolerance : ' and the user enters '0.1'. The program then outputs '4.3125'.

# Chapter 5: Fixed-Point Method

## Implementation

- `phi_function(x)`: Computes the value of the function  $\varphi(x)$  at a given point  $x_n$ , following the recurrence relation:

$$x_{n+1} = \varphi(x_n)$$

- `ErrorEstimation(a, b)`: Estimates the error at iteration  $n$  for a given contraction factor  $k \in ]0, 1[$ , using the formula:

$$\frac{k^n}{1-k} \cdot |x_0 - x_1|$$

- `eps`: Defines the error tolerance  $\epsilon$ , determining the stopping criterion.
- `max_iter`: Specifies the maximum number of iterations allowed in the fixed-point algorithm.
- `Fixed_Point(eps, k, x_0, function, max_iter=100)`: Computes the root of  $f(x)$  with the fixed-point method with the its corresponding  $\varphi(x)$  and contraction factor  $k$ , starting from an initial  $x_0$  with a tolerance  $\epsilon$  and the maximum number of iterations `max_iter` (default: 100).

## Code

```
1 def ErrorEstimation(x_0,x_1,k,n):
2     return k**(n)/(1-k) * abs(x_0-x_1)
3
4 def phi_function(x):
5     return # function
6
7
8 def Fixed_Point(eps,k,x_0,phi_function,max_iter=100):
9
10    x_1 = x_n = phi_function(x_0)
11
12    n = 0
13
14    while (error:= ErrorEstimation(x_0,x_1,k,n)) > eps and n<= max_iter:
15        x_n = phi_function(x_n)
16        n = n+1
17
18    return x_n
19
20 eps = value_1
21 x_0 = value_2
22
23 print(Fixed_Point(eps,k,phi_function))
```

## Example

```
1 import numpy as np
2
3 def ErrorEstimation(x_0,x_1,k,n):
4     return k**(n)/(1-k) * abs(x_0-x_1)
5
6 def phi_function(x):
7     return x*np.exp(x)/3 - 1
8
9
10 def Fixed_Point(eps,k,x_0,phi_function,max_iter=100):
11
12     x_1 = x_n = phi_function(x_0)
13
14     n = 0
15
16     while (error:= ErrorEstimation(x_0,x_1,k,n)) > eps and n<= max_iter:
17         x_n = phi_function(x_n)
18         n = n+1
19
20     return x_n
21
22 eps = 10**(-2)
23 x_0 = -0.5
24 k = 0.045
25 print(Fixed_Point(eps,k,x_0,phi_function))
```

```
rabah@UbuntuTex: ~/Desktop/Documentations/University/3rd_IN
python/Chapters/Code/FIX$ python3 ex1.py
-1.1217842921577577
```