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`.

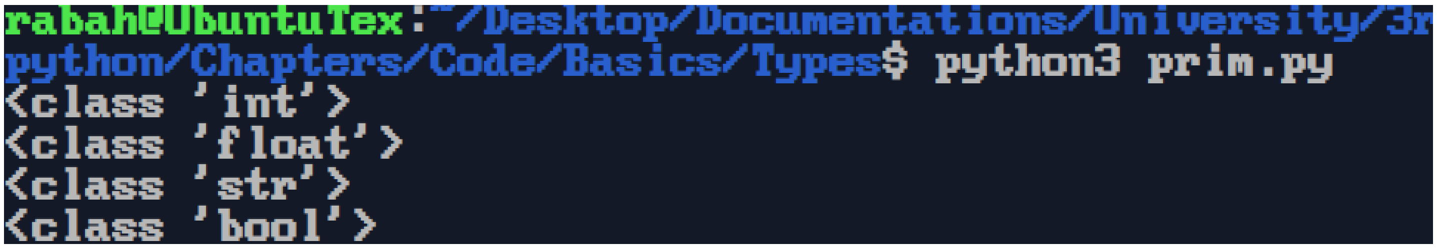
2.2 Non-Primitive Types

Non-Primitive Types

- **list**: Stores an ordered collection of values.
- **tuple**: Similar to a list but we can't change their value directly only overide 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.

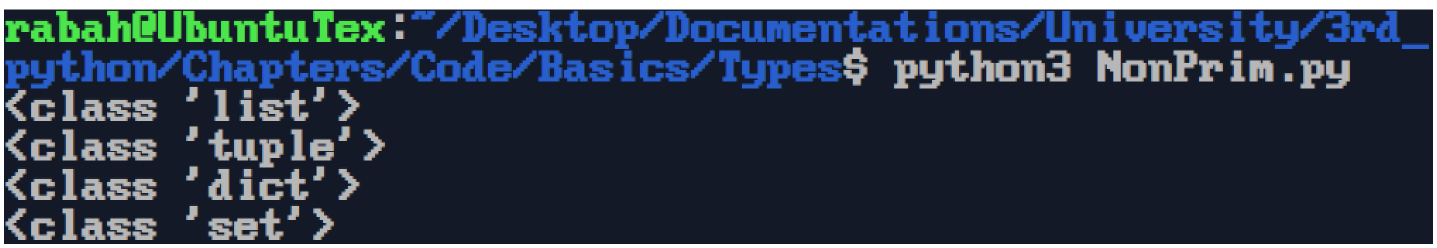
Primitive Example

```python
age = 20
salary = 45532.21
name = "latex"
isTired = True

print(type(age))
print(type(salary))
print(type(name))
print(type(isTired))
```



Non Primitive Example

```python
gradeList = [19, 16, 12, 11]
coordinatesTuple = (1, 2.5)
personMap = {"id": "1879", "data": "404 not found"}
uniqueNbSet = {1, 2, 3, 4}

print(type(gradeList))
print(type(coordinatesTuple))
print(type(personMap))
print(type(uniqueNbSet))
```



3 Comments

Comments

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

Example

```python
## this is a single comment
print("hello world")

'''
This is a
 block comment
'''

"""
This is another
 block comment
"""
```

4 Input/Output

IO

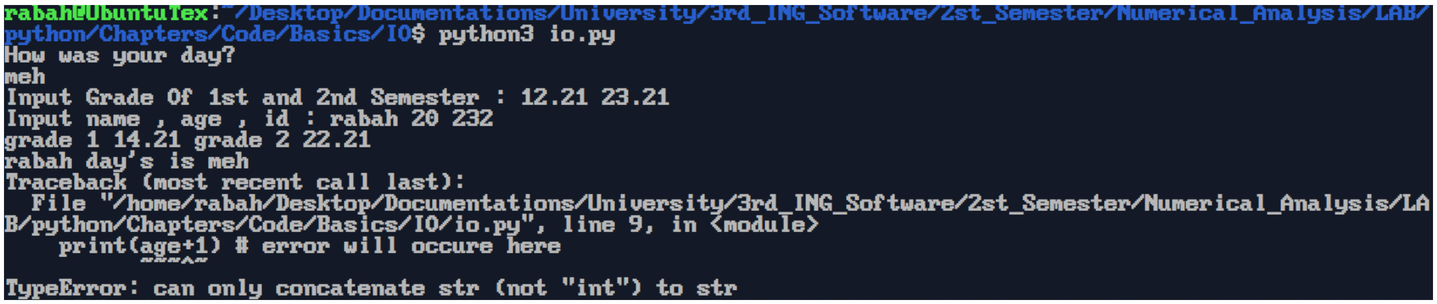
- `print` : we use the print function to display text , there are many string formatting we will see them in next section , the default formating is text between qout and we seprate variable with comma , and it automatically add space
- `input` : has message inside ,by default input takes in string variable.we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

Syntax

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

Example

```python
hru = input("How was your day?\n") #no need for casting

grade1,grade2 = map(float,input("Input Grade Of 1st and 2nd Semester : ").split()) # input 2 float we must type cast
name , age , id = input("Input name , age , id : ").split() # if we try arithmetic operation on age we will get error
                                                               # not casted to int

print('grade 1',grade1+2 ,'grade 2',grade2-1)
print(name,'day\'s is',hru)
print(age+1) # error will occure here
```



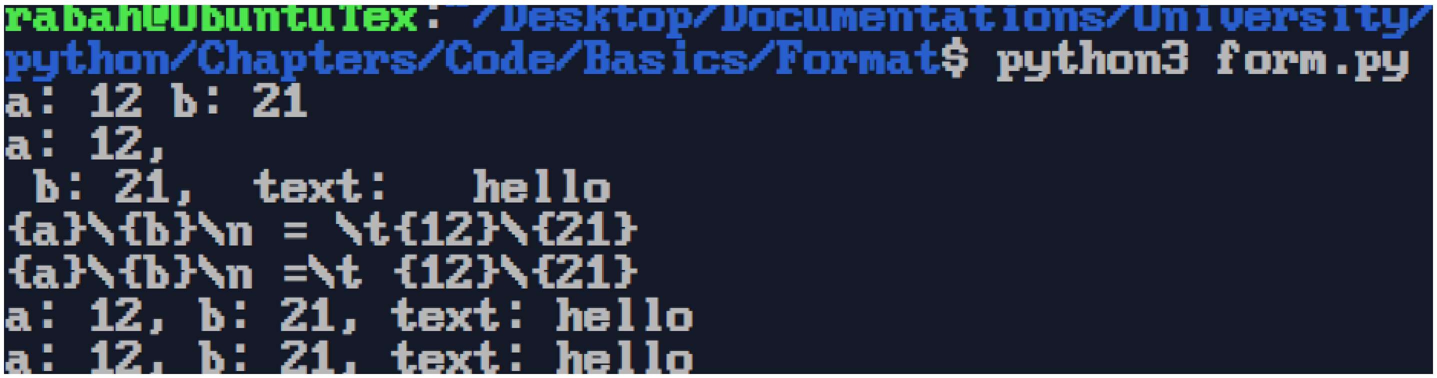
5 String Formatting

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

```python
a = 12
b = 21
text = "hello"

print("a:", a, "b:", b)                            # default formatting

print(f"a: {a},\n b: {b},\t text:\t {text}")       # f-string formatting

print(r"{a}\{b}\n = \t{12}\{21}")                  # raw-string formatting

print(fr"{{a}}\{{b}}\n =\t {{{a}}}\{{{b}}}")       # f+raw-string formatting

print("a: %d, b: %d, text: %s" % (a, b, text))     # %-formattting

print("a: {}, b: {}, text: {}".format(a, b, text)) # .format()
```



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

```python
# Note as are optional

import moduleName as moduleAlias  # import the whole module
from moduleName import Part_1 as PartAlias_1 , ... , Part_n as PartAlias_n  #import parts of the module
```

Example

```python
import datetime as dt                 #import the whole datetime module as dt
import cmath                          #import the whole module cmath
import math                           #import the whole module math
from math import sqrt as sq , pi      #import sqrt function as sq and pi variable from math module
from cmath import sqrt as sq          #import sqrt function as sq from cmath module

print(dt.datetime.now())              #call the datetime.now() function of dt
print(cmath.sqrt(5))                  #call the sqrt() function of camth
print(pi)                             #call the pi variable of math
print(math.sqrt(5))                   #call the sqrt() function of math
print(sq(5))                          #call the sq() function of cmath (it was overriden 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

```
if condition:
    # instructions indented with 2 spaces
elif condition:
    # instructions indented with 2 spaces
else:
    # instructions indented with 2 spaces
```

Example

```
a = 0;

if a > 0:
    print(a,"is strictly positive")
elif a < 0:
    print(a,"is strictly negative")
else:
    print(a,"is null")
```

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

8.2 Match Statement(Switch Case)

Syntax

```
match var:
    case value_1:
        # instructions idented with 8 spaces

    ............

    case value_n:
        # instructions indented with 8 spaces

    case default: # default if var isn't in (value_1,...,value_n)
        # instructions indented with 8 spaces
```

Example

```
a = 20

match a:
    case -21:
        print("negative")
    case 2|10|21:
        print("positive")
    case int() :
        print("integer")
    case default :
        print("idk")
```

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

8.3 While Loop

Syntax

```python
while condition:
    #instructions indented with 4 spaces
```

Example

```python
cpt = 5

while cpt > 0:
    print(cpt)
    cpt = cpt - 1;
```

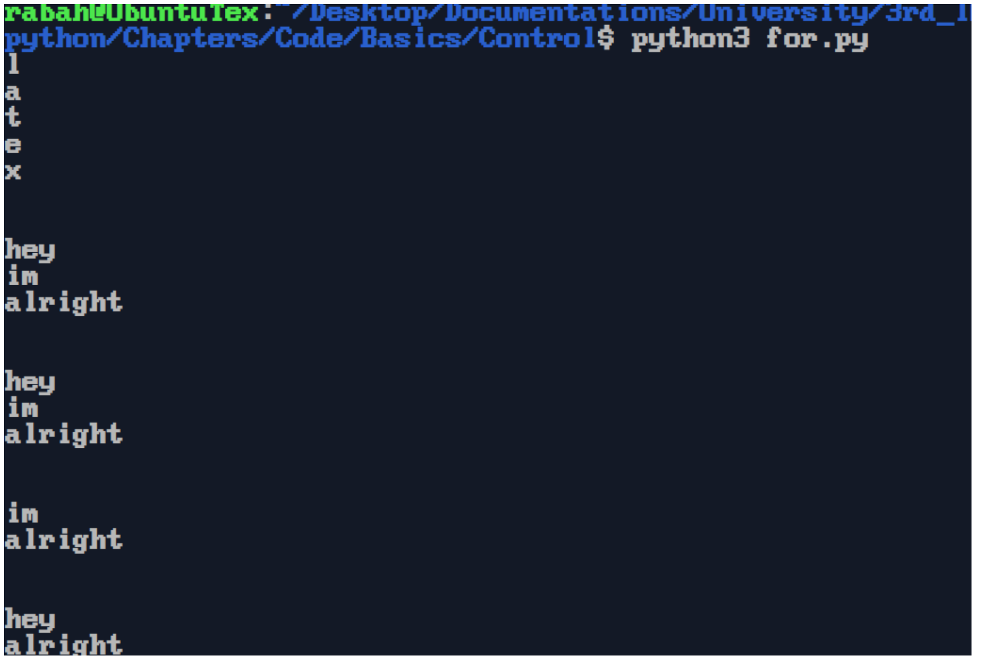
8.4 For Loop

Syntax

```python
for value in iterable:  # Iterates over each item in an iterable (e.g., list, tuple, set, string, generator)
    # Instructions indented with 4 spaces

for index in range(n):  # Loops from 0 to (n-1), incrementing by 1
    # Instructions indented with 4 spaces

for index in range(a, n):  # Loops from a to (n-1), incrementing by 1
    # Instructions indented with 4 spaces

for index in range(a, n, b):  # Loops from a to (n-1), incrementing by b (can be negative for reverse loops)
    # Instructions indented with 4 spaces
```

Example

```
for value in "latex": # ['l','a','t','e','x']
    print(value)

strList = ["hey","im","alright"]

print("\n")

for value in strList: # ["hey","im","alright"]
    print(value)

print("\n")

n = len(strList)

for i in range(n): # ["hey","im","alright"]
    print(strList[i])

print("\n")

for i in range(1,n): # ["im","alright"]
    print(strList[i])

print("\n")

for i in range(0,n,2): # ["hey","alright"]
    print(strList[i])
```



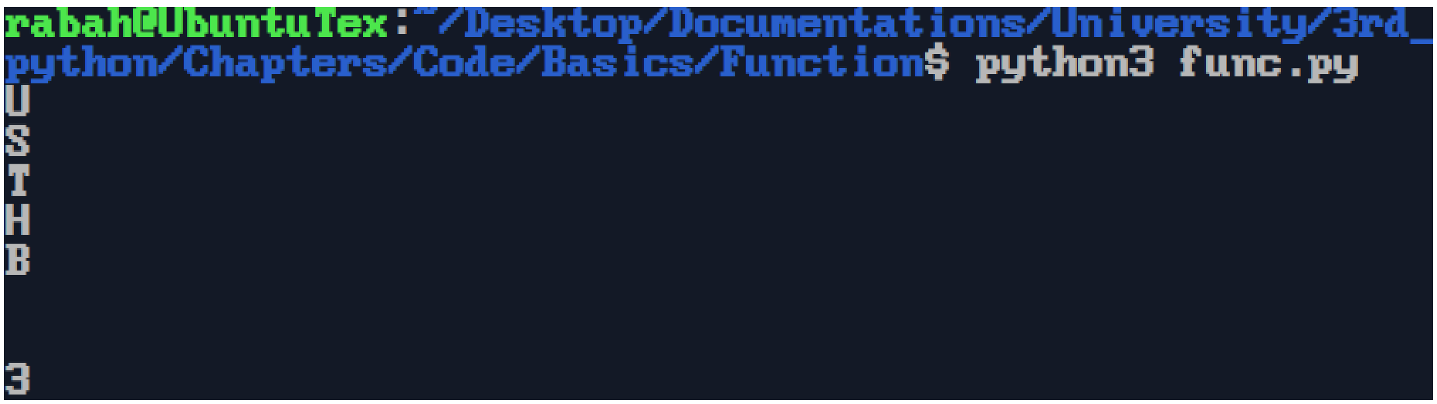
9 Function

Syntax

```
def function_name():
    # Instructions indented with 4 spaces
    return value

def procedure_name():
    # Instructions indented with 4 spaces
```

Example

```
def sum(a,b):
    return a+b

def print_char(string):
    for char in string:
        print(char)

    print("\n")

print_char("USTHB")
s = sum(1,2)
print(s)
```



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

```python
from collections import namedtuple

# Defining a NamedTuple with its name and attributes
StructName = namedtuple("StructName", ["attributeName_1",...,"attributeName_n"])

# Create an instance of the NamedTuple
struct_var = StructName(value_1,..., value_n)

# Accessing attributes with .
print(struct_var.attributeName_n)
```

Example

```python
from collections import namedtuple

intervalle = namedtuple("intervalle", ["a","b"])

intervalle_var = intervalle(1,2)

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 the array() function.

Syntax

```
import numpy as np

npArray = np.array(List)
```

Example

```
import numpy as np

List = [1,2,3]

npArray_1 = np.array(List)
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.

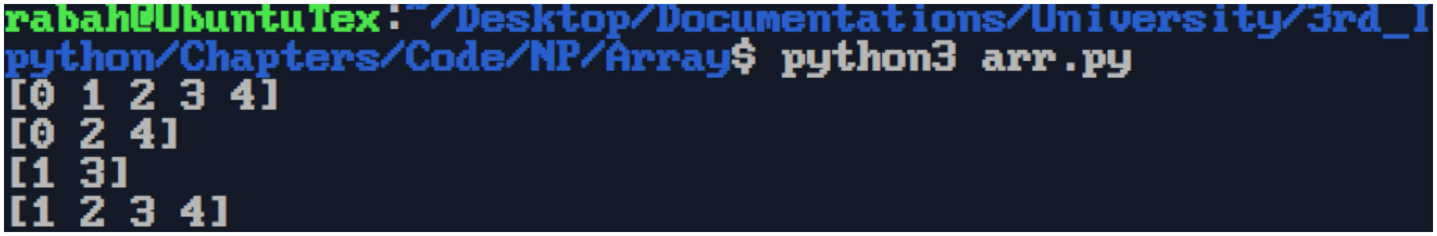
Arange()

Syntax

```
import numpy as np

Array_1 = np.arange(n)              # generate array from 0 to (n-1) with step = 1
Array_2 = np.arange(a, n)           # generate array from a to (n-1) with step = 1
Array_3 = np.arange(a, n, b)        # generate array from a to less than n with step = b
Array_4 = np.arange(stop=n, step=b) # generate array from 0 to less than n with step = b
```

Example

```
import numpy as np

array_1  = np.arange(5)
array_2  = np.arange(stop=5,step=2)
array_3  = np.arange(1,5,2)
array_4  = np.arange(1,5)

print(array_1)
print(array_2)
print(array_3)
print(array_4)
```



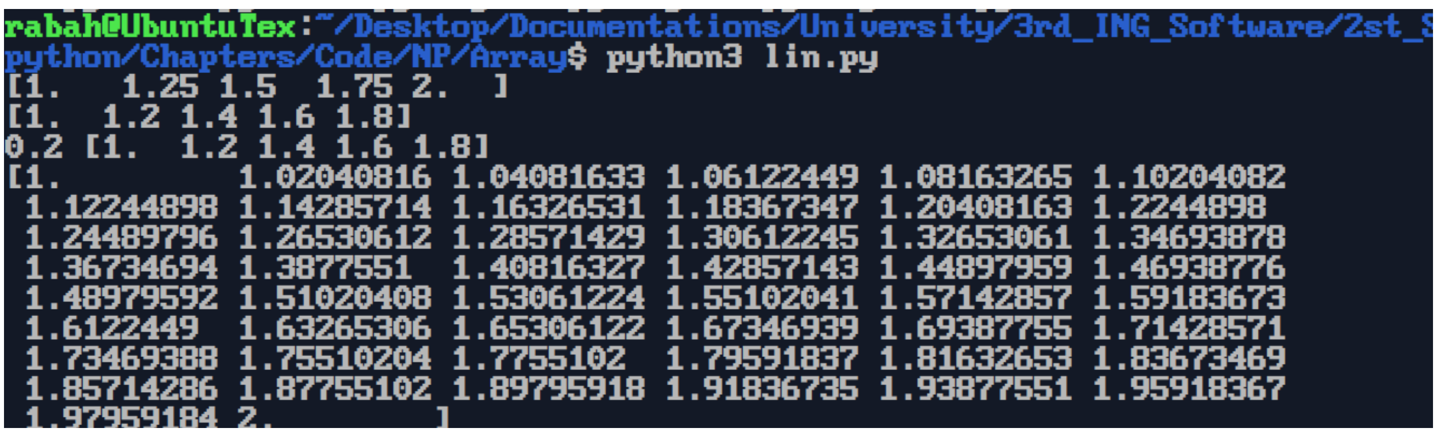
LinSpace()

Syntax

```
import numpy as np

Array_1 = np.linspace(a, n)           # generate array from a to n with 50 elements (default)
Array_2 = np.linspace(a, n, b)        # generate array from a to n with b elements
Array_3 = np.linspace(a, n, b, False) # generate array from a to n (endpoint excluded) with b elements
Array_4, step = np.linspace(a, n, b, False, True) # generate array from a to n (endpoint excluded) with b elements
                                                   # also returns step size
```

Example

```
import numpy as np

array_1  = np.linspace(1,2,5)
array_2 = np.linspace(1,2, 5, False)
array_3, step = np.linspace(1, 2, 5, False, True)
array_4  = np.linspace(1,2)

print(array_1)
print(array_2)
print(step,array_3)
print(array_4)
```



2.3 Array Slicing

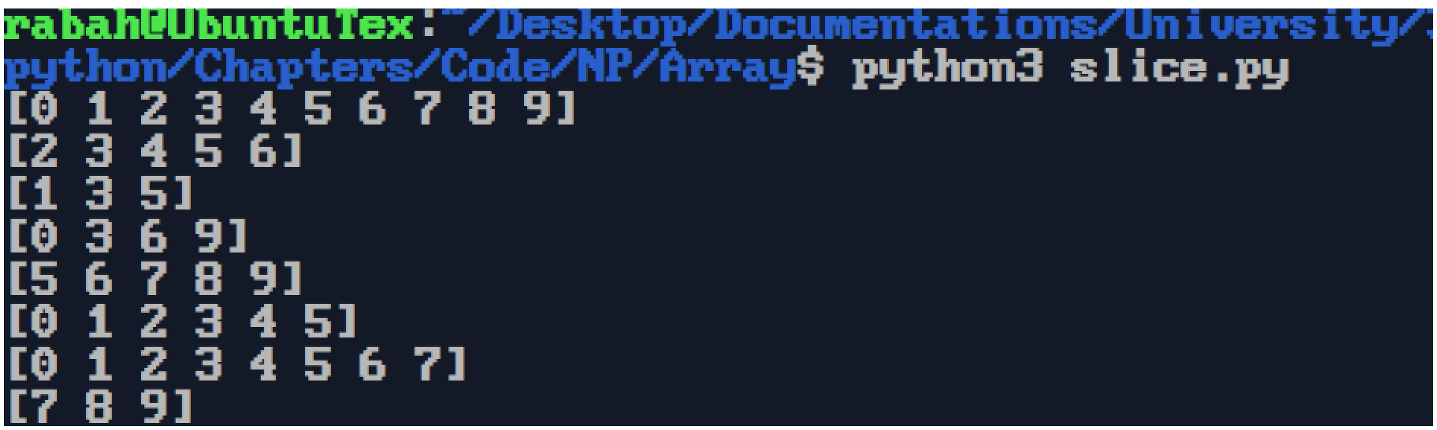
Slicing

`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

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

print(arr[::])
print(arr[2:7])
print(arr[1:7:2])
print(arr[::3])
print(arr[5:])
print(arr[:6])
print(arr[:-2])
print(arr[-3:])
```



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

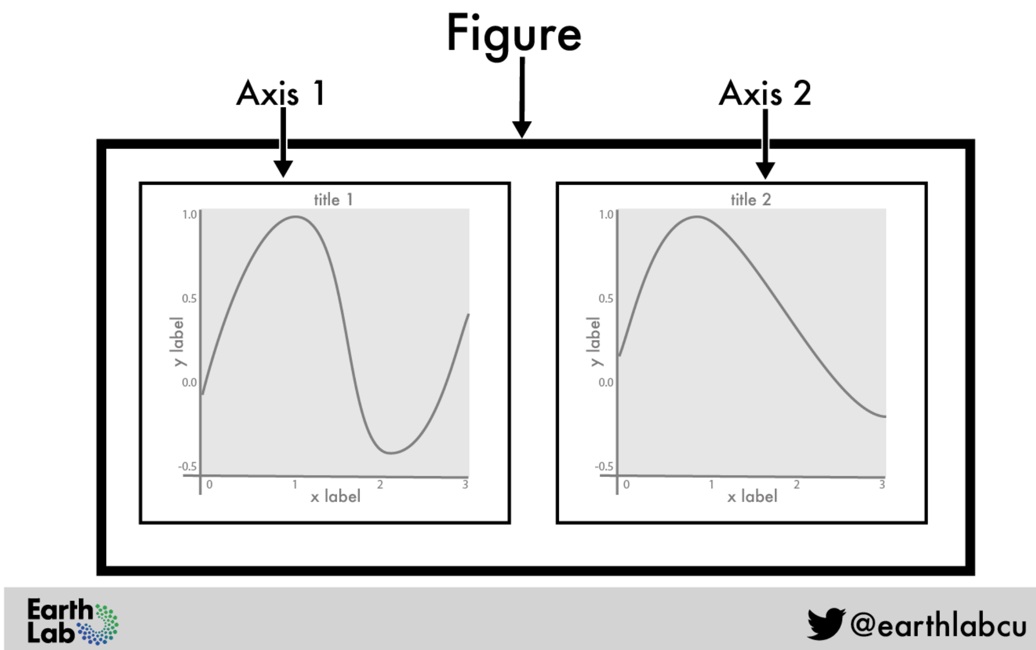
Introduction

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 + (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	χ
'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

```
import matplotlib.pyplot as plt
import numpy as np

x1 = np.linspace(-2,2,100)
x2 = np.linspace(0.5,5,100)
x3 = np.linspace(-np.pi,np.pi,100)

y1 = np.exp(x1)
y2 = np.log(x2)
y3 = np.cos(x3)

plt.subplot(1,2,1)
plt.plot(x1,y1,label=r"$f_1(x) = e^x$",color = "red",linestyle='--',linewidth=1.75)
plt.title(r"$f_1(x)$")
plt.xlabel(r"$x_1$")
plt.ylabel(r"$y_1$")
plt.plot(np.array([0,1]),np.array([1,np.exp(1)]),linestyle='',color="lightblue",label="Points",marker='o')
plt.grid()
plt.legend()

plt.subplot(1,2,2)
plt.plot(x2,y2,label=r"$f_2(x) = \ln{(x)}$",color = "blue")
plt.title(r"$f_2(x)$")
plt.scatter(1,0,color="lightblue",label=r"$\alpha_1$",marker='*')
plt.scatter(2,np.log(2),color="lightgreen",label=r"$\alpha_2$",marker='D')
plt.legend()

plt.suptitle(r"figure$_1$")
plt.savefig("fig1.pdf")


plt.figure()

plt.plot(x3,y3,label=r"$f_3(x) = \cos{(x)}$",color = "green",linewidth=1.25)
plt.title(r"$f_3(x)$")
plt.grid(True)

plt.suptitle(r"figure$_2$")
plt.savefig("fig2.pdf")

plt.show()
```

fig1

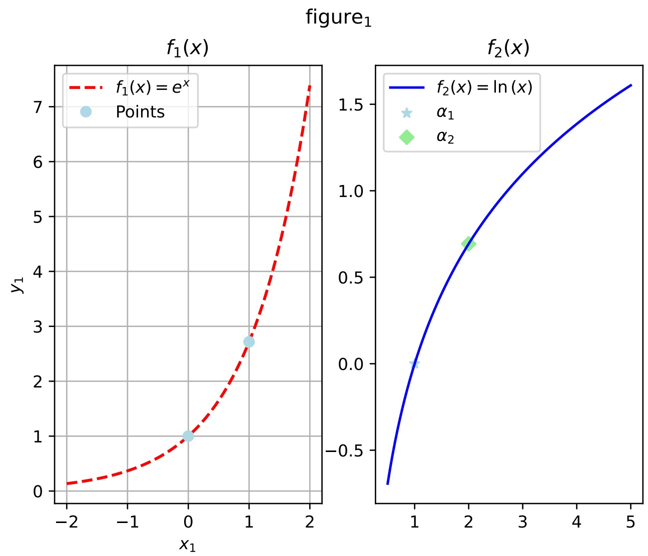
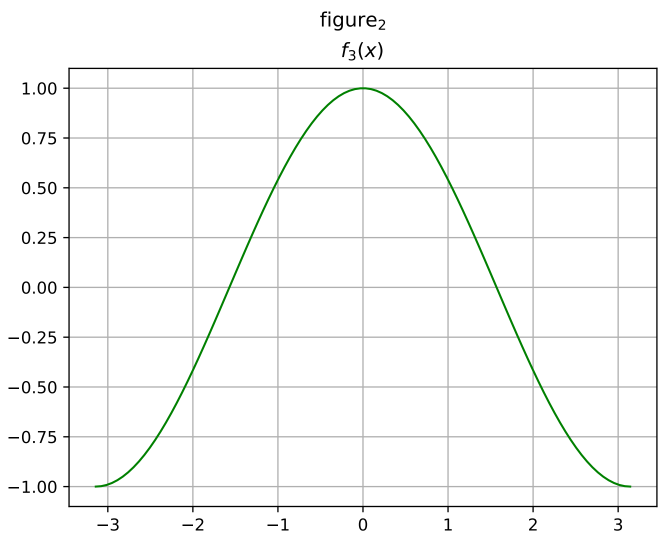

figure$_1$
$f_1(x)$
$f_2(x)$
$f_1(x) = e^x$
Points
$f_2(x) = \ln(x)$
α_1
α_2
y_1
x_1
7
6
5
4
3
2
1
0
−2
−1
0
1
2
1.5
1.0
0.5
0.0
−0.5
1
2
3
4
5

fig 2


figure$_2$
$f_3(x)$
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−3
−2
−1
0
1
2
3

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.

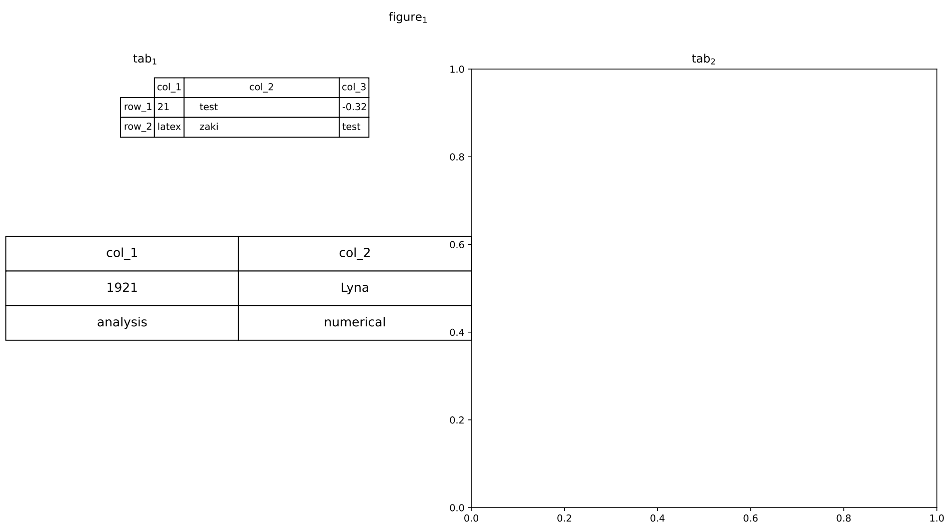
Some Function of `Table` object

- `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

```python
import matplotlib.pyplot as plt
import numpy as np

plt.subplot(1,2,1)
cellText_1 = [["21","test","-0.32"] , ["latex" , "zaki" , "test"]]
rowLabels_1 = ["row_1","row_2"]
colLabels_1 = ["col_1","col_2","col_3"]
table_1 = plt.table(fontsize=12,cellText=cellText_1,rowLabels=rowLabels_1,colLabels=colLabels_1,cellLoc = 'left',loc='best')
table_1.auto_set_column_width ([0,2])
plt.axis('off')
plt.title(r"tab$_1$")

plt.subplot(1,2,2)
cellText_2 = [["1921","Lyna"] , ["analysis" , "numerical"]]
colLabels_2 = ["col_1","col_2"]
table_2 = plt.table(cellText=cellText_2,colLabels=colLabels_2,cellLoc = 'center',loc='left')
table_2.scale(xscale = 1 , yscale = 1.75)
table_2.set_fontsize(40)
table_2.auto_set_font_size()

plt.title(r"tab$_2$")

plt.suptitle(r"figure$_1$")

plt.show()
```

fig1



Chapter 4: Dichotomy Mehtod

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 ϵ.
- `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 ϵ , and max number of iteration (100 by default).

Code

```
def function(x):
    return ## function

def ErrorEstimation(a,b):
    return (b-a)/2


def dichotomy(eps,a,b,function,max_iter=100):

    n = 1

    while ErrorEstimation(a,b) > eps and n <=max_iter :

        x = (a+b)/2

        if function(x) * function(a) < 0:
            b = x
        elif function(x) * function(b) < 0:
            a = x
        else:
            return x

        n = n+1

    return (a+b)/2



eps = value
a = value_1
b = value_2
print(dichotomy(eps,a,b,function))
```

Example

```python
def function(x):
    return x**3 - 80

def ErrorEstimation(a,b):
    return (b-a)/2


def dichotomy(eps,a,b,function,max_iter=100):

    n = 1

    while ErrorEstimation(a,b) > eps and n <=max_iter :

        x = (a+b)/2

        if function(x) * function(a) < 0:
            b = x
        elif function(x) * function(b) < 0:
            a = x
        else:
            return x

        n = n+1

    return (a+b)/2


eps = float(input("Input The Tolerance : "))
a = 4
b = 5
print(dichotomy(eps,a,b,function))
```

```
rabah@UbuntuTex:~/Desktop/Documentations/University
python/Chapters/Code/DIC$ python3 ex1.py
Input The Tolerance : 0.1
4.3125
```

Chapter 5: Fixed-Point Mehtod

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 ϵ, 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 ϵ and the maximum number of iterations max_iter (default: 100).

Code

```
def ErrorEstimation(x_0,x_1,k,n):
    return k**(n)/(1-k) * abs(x_0-x_1)

def phi_function(x):
    return  # function


def Fixed_Point(eps,k,x_0,phi_function,max_iter=100):

    x_1 = x_n = phi_function(x_0)

    n = 0

    while (error:= ErrotEstimation(x_0,x_1,k,n) > eps and n<= max_iter):
        x_n = phi_function(x_n)
        n = n+1

    return x_n

eps = value_1
x_0 = value_2

print(Fixed_Point(eps,k,phi_function))
```

Example

```python
import numpy as np

def ErrorEstimation(x_0,x_1,k,n):
    return k**(n)/(1-k) * abs(x_0-x_1)

def phi_function(x):
    return x*np.exp(x)/3 - 1


def Fixed_Point(eps,k,x_0,phi_function,max_iter=100):

    x_1 = x_n = phi_function(x_0)

    n = 0

    while (error:= ErrorEstimation(x_0,x_1,k,n) > eps and n<= max_iter):
        x_n = phi_function(x_n)
        n = n+1

    return x_n

eps = 10**(-2)
x_0 = -0.5
k = 0.045
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
```