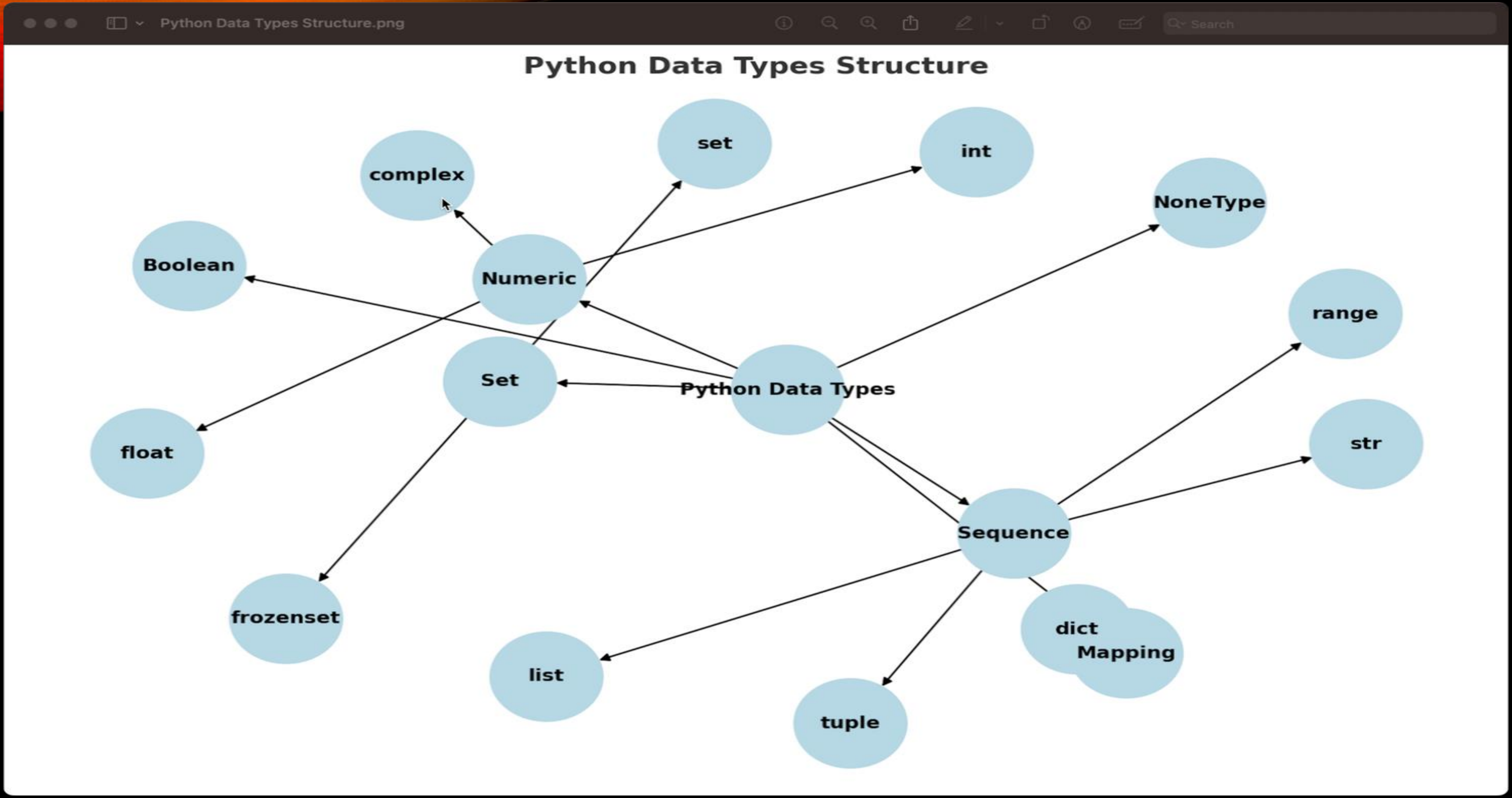
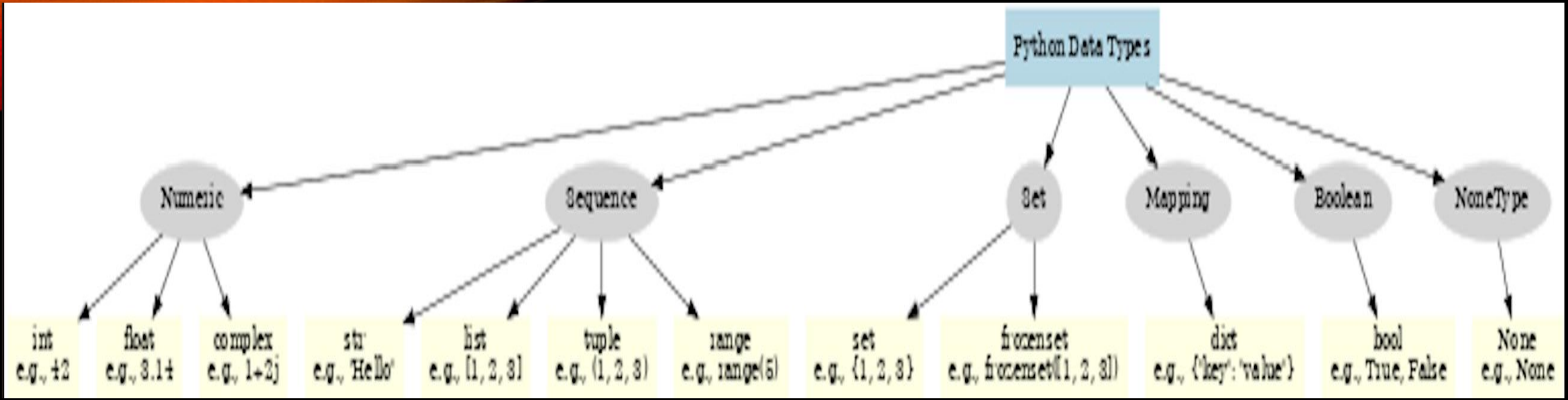
PYTHON – BASIC COURSE

WHY PYTHON FOR DATA SCIENCE

- Python has been a cross-platform, open-source language for over 20 years. You can code seamlessly on **Linux**, **Windows**, and **MacOS**. Its versatility and ease of use make it an excellent choice for data-related work.
- **Benefits of Python:**
- 1. **Reduces Development Time**
- Python is efficient and helps you build projects faster.
- 2. **Object-Oriented Language**
- It supports object-oriented programming, simplifying complex applications.
- 3. **No Compilation Required**
- Python runs without manual compilation, saving time.
- 4. **Dynamic Data Typing**
- It adapts to different data types during runtime.
- 5. **Shorter Code**
- Python allows you to achieve more with fewer lines of code.
- 6. **Easy to Learn**
- Python’s simplicity makes it beginner-friendly and developer-friendly.
- 7. **Readable Code**
- Python code is clean and easy to understand, even for teams.
- 8. **Supports Collaboration**
- Ideal for team projects due to its simplicity and readability.
- 9. **Extendable**
- You can easily integrate Python with other programming languages.
- 10. **Automatic Memory Management**
- Python handles memory allocation and deallocation automatically.
- 11. **Free and Open Source**
- Python is completely free to use and modify.

Data Types of Python





Python Data Types

Numeric

Sequence

Set

Mapping

Boolean

NoneType

int
e.g., 42

float
e.g., 3.14

complex
e.g., 1+2j

str
e.g., 'Hello'

list
e.g., [1, 2, 3]

tuple
e.g., (1, 2, 3)

range
e.g., range(5)

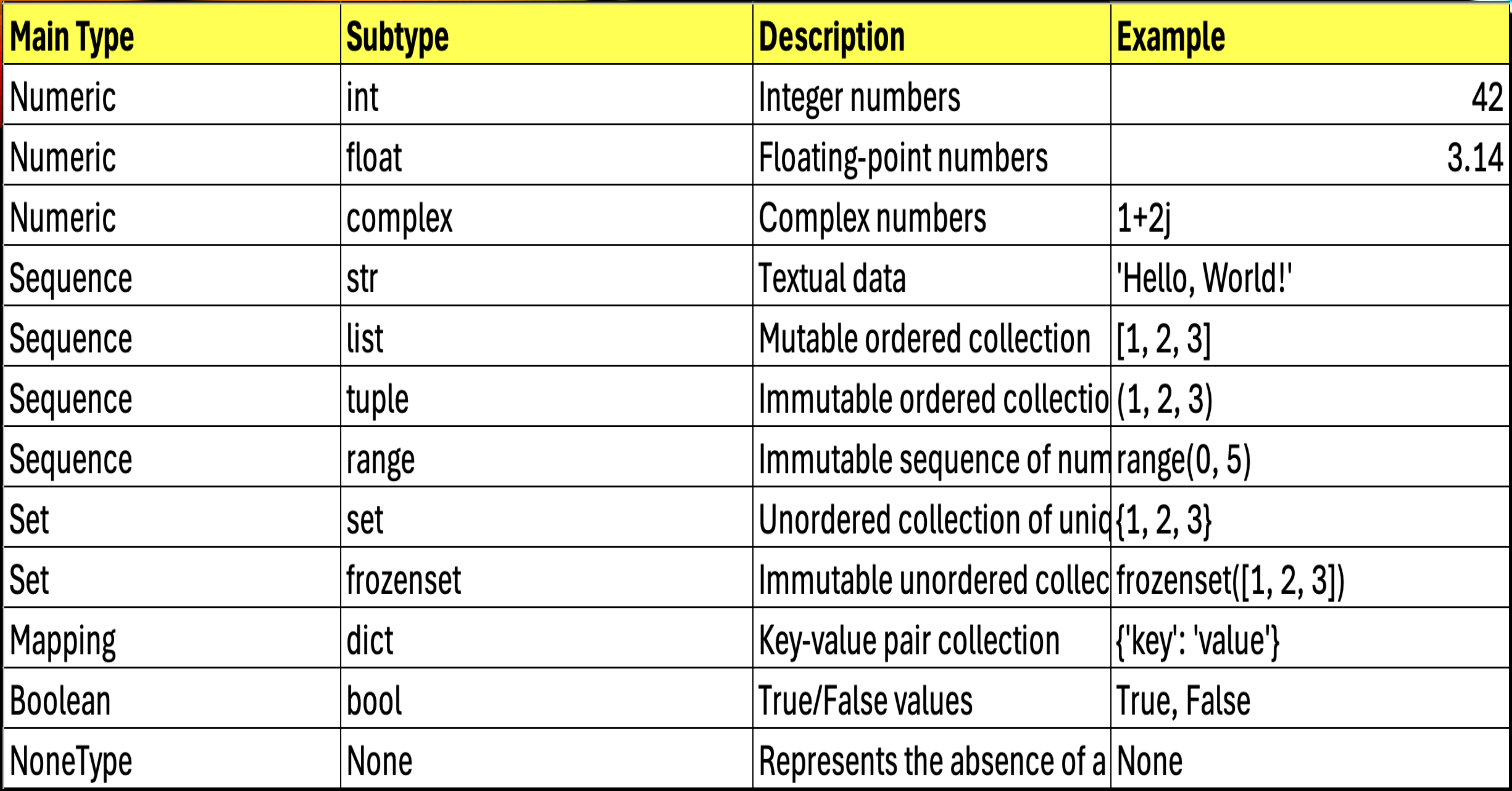
set
e.g., {1, 2, 3}

frozenset
e.g., frozenset([1, 2, 3])

dict
e.g., {'key': 'value'}

bool
e.g., True, False

None
e.g., None



Main Type	Subtype	Description	Example
Numeric	int	Integer numbers	42
Numeric	float	Floating-point numbers	3.14
Numeric	complex	Complex numbers	1+2j
Sequence	str	Textual data	'Hello, World!'
Sequence	list	Mutable ordered collection	[1, 2, 3]
Sequence	tuple	Immutable ordered collectio	(1, 2, 3)
Sequence	range	Immutable sequence of num	range(0, 5)
Set	set	Unordered collection of uniq	{1, 2, 3}
Set	frozenset	Immutable unordered collec	frozenset([1, 2, 3])
Mapping	dict	Key-value pair collection	{'key': 'value'}
Boolean	bool	True/False values	True, False
NoneType	None	Represents the absence of a	None

Strings : A basic sequence of characters or basically a text. Strings are groups of letters and/or characters delimited with quotation marks, single or double. Strings are amongst the most popular types in Python. There are a number of methods or built-in string functions

Defining Strings :

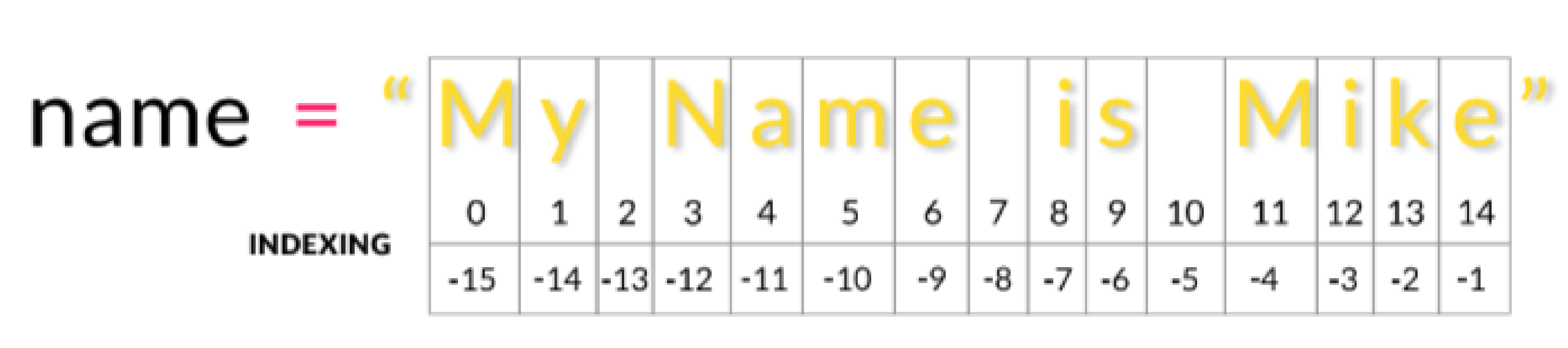
- name=”Python"
- print(name)

Accessing strings :

- print(name[0])
- print(name[0:3])
- print(name[3:6])
- print(name[3:4])

String Operations :

- name=”Python”   len(name)
- name.upper()
- name.lower()
- name.title()   - -> Converts first letter of each word into capital
- Name.count() → count number of times repletion of each word
- Name.index() → Starting index of given word’s index starting from zero
- Name.replace(‘is’, ‘was’) → replace word with new word
- Name.split() → split given string
- Name.join() → join given words



name = “	M	y		N	a	m	e		i	s		M	i	k	e	”
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
INDEXING	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	

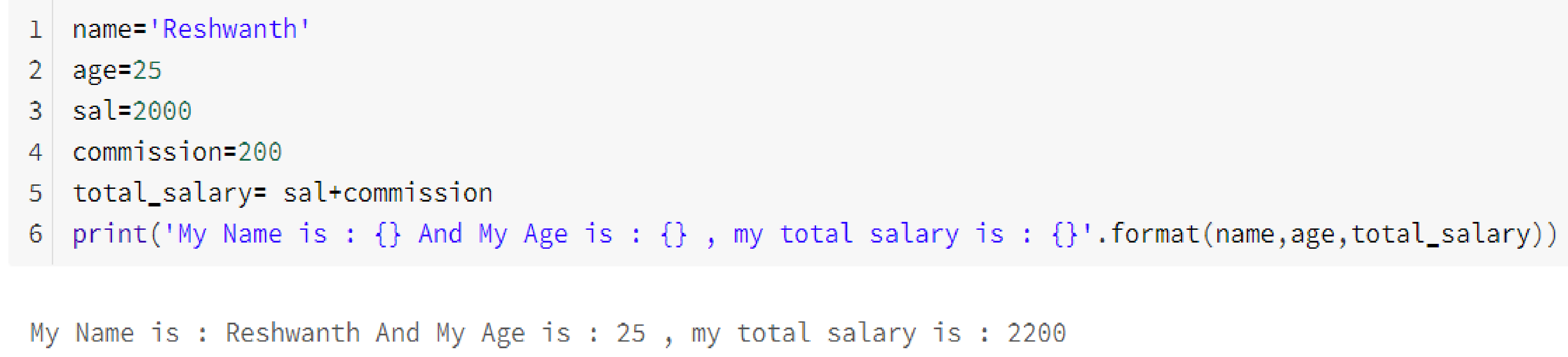
F-strings

F-Strings provide a way to embed expressions inside string literals, using a minimal syntax. It should be noted that an f-string is really an expression evaluated at run time, not a constant value. In Python source code, an f-string is a literal string, prefixed with 'f', which contains expressions inside braces. The expressions are replaced with their values.

```
Name = ‘Python’
Salary = 100000.74
f_string = f ‘my name is ‘ :{name} and my age is :{age}’
Print(f_string)
```

format() function

- str.format()` is one of the string formatting methods in Python3, which allows multiple substitutions and value formatting. This method lets us concatenate elements within a string through positional formatting.
- * __Syntax__ : `{ } .format(value)`
- Parameters : `(value)` : Can be an integer, floating point numeric constant, string, characters or even variables .Returntype : Returns a formatted string with the value passed as parameter in the placeholder position. The placeholders can be identified using named indexes {price}, numbered indexes {0}, or even empty placeholders {}



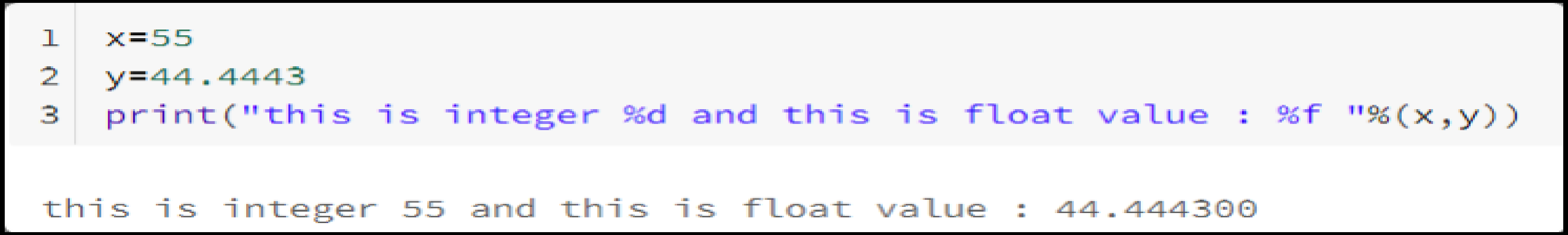
```python
name='Reshwanth'
age=25
sal=2000
commission=200
total_salary= sal+commission
print('My Name is : {} And My Age is : {} , my total salary is : {}'.format(name,age,total_salary))
```

```
My Name is : Reshwanth And My Age is : 25 , my total salary is : 2200
```

String Formatting

Python uses C-style string formatting to create new, formatted strings. The "%" operator is used to format a set of variables enclosed in a "tuple" (a fixed size list), together with a format string, which contains normal text together with "argument specifiers", special symbols like "%s" and "%d".

- s – strings
- d – decimal integers (base-10)
- f – floating point display
- c – character
- b – binary
- o – octal
- x – hexadecimal with lowercase letters after 9
- X – hexadecimal with uppercase letters after 9
- e – exponent notation



```python
x=55
y=44.4443
print("this is integer %d and this is float value : %f "%(x,y))
```

```
this is integer 55 and this is float value : 44.444300
```

BOOLEANS

- Booleans are probably the most simple data type in Python. They can only have
- one of two values, namely True or False. It’s a binary data type. We will use it a
- lot when we get to conditions and loops. The keyword here is bool.

LIST []

- Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets []. Lists might contain items of different types, but usually the items all have the same type.

➢ List is a collection which is ordered and changeable. Allows duplicate members.

➢ In Python lists are written with square brackets.

➢ Note: Python Lists replace Arrays (from most programming languages)

- numbers = [10, 22, 6, 1, 29]

In Python, we define lists by using square brackets. We put the elements in between of those and separate them by commas. The elements of a list can have any data type and we can also mix them.

- numbers = [10, 22, 6, 1, 29]
- names = ["John", "Alex", "Bob"]
- mixed = ["Anna", 20, 28.12, True]

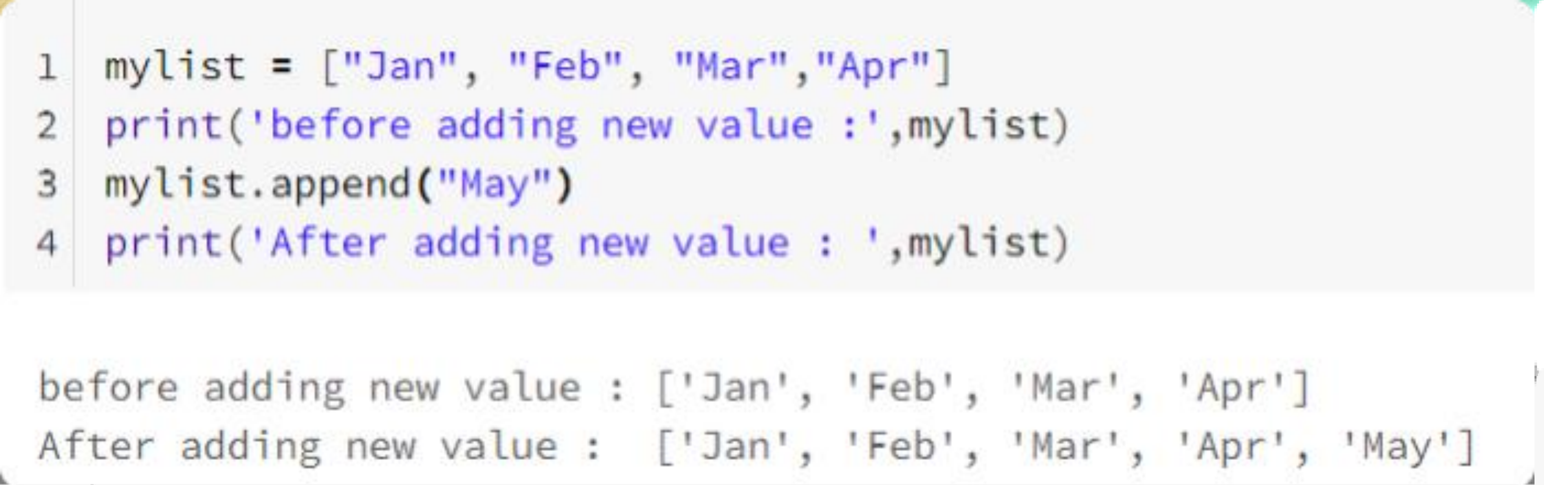
- print(numbers[2])
- print(mixed[1])
- print(names[0])

In a list, we can also modify the values. For this, we index the elements in the same way.

- numbers[1] = 10
- names[2] = "Jack”

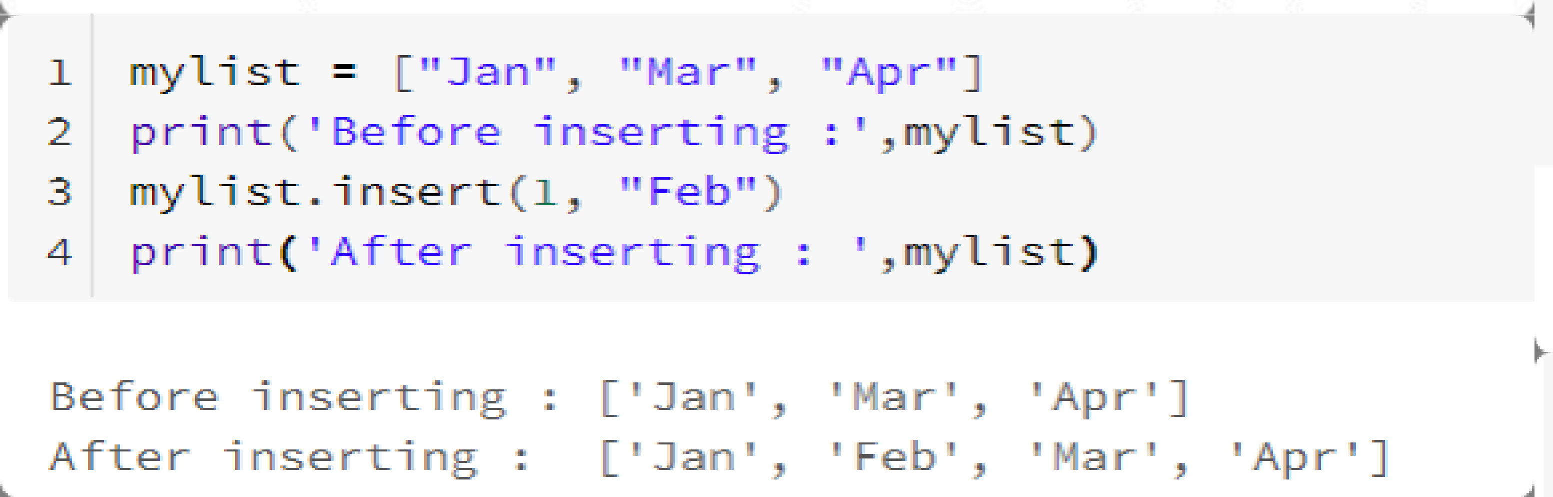
Other method of list are as below

- len(variable_name)
- Sum()
- Max()
- Index()
- Count()
- Reverse()
- Sort(), sort(reverse=True)
- Append(), insert(), remove(), pop(), copy(),del(),extend()



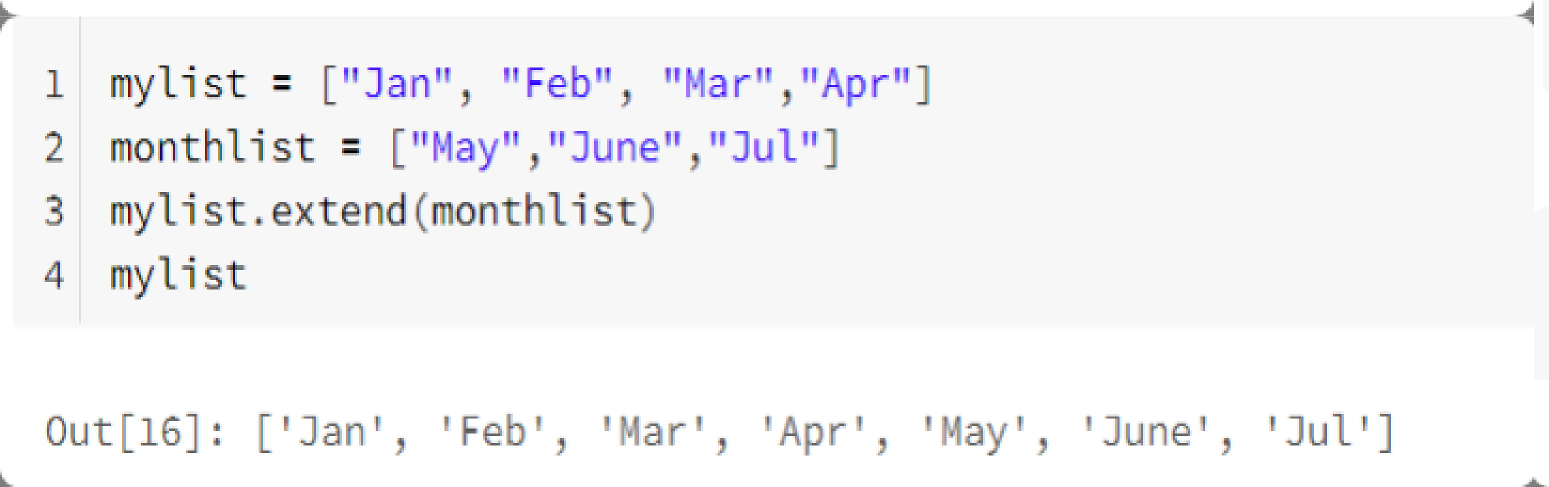
```
mylist = ["Jan", "Feb", "Mar","Apr"]
print('before adding new value :',mylist)
mylist.append("May")
print('After adding new value : ',mylist)
```

```
before adding new value : ['Jan', 'Feb', 'Mar', 'Apr']
After adding new value :  ['Jan', 'Feb', 'Mar', 'Apr', 'May']
```



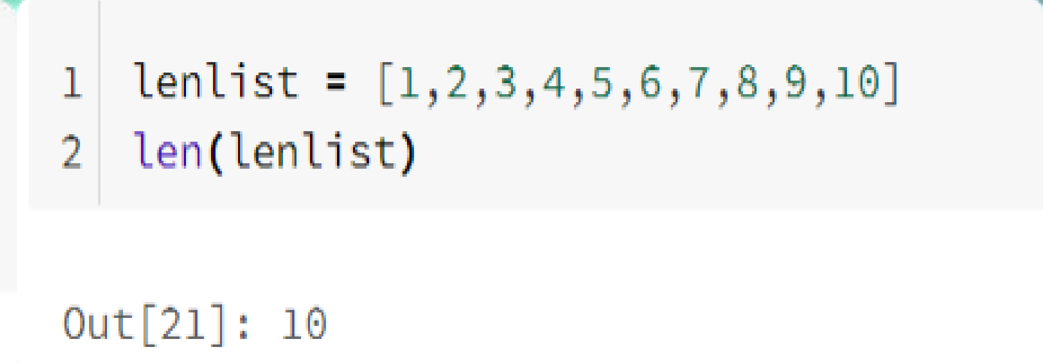
```
mylist = ["Jan", "Mar", "Apr"]
print('Before inserting :',mylist)
mylist.insert(1, "Feb")
print('After inserting : ',mylist)
```

```
Before inserting : ['Jan', 'Mar', 'Apr']
After inserting :  ['Jan', 'Feb', 'Mar', 'Apr']
```



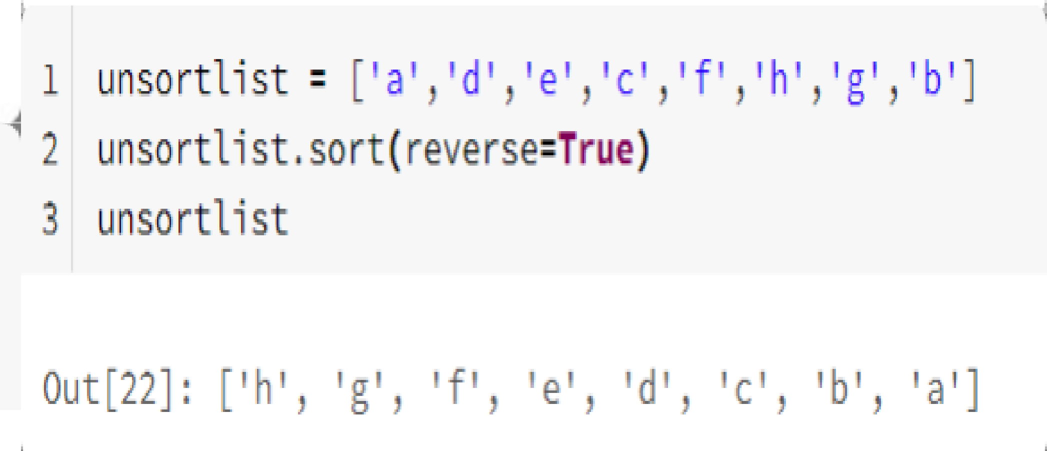
```
mylist = ["Jan", "Feb", "Mar","Apr"]
monthlist = ["May","June","Jul"]
mylist.extend(monthlist)
mylist
```

```
Out[16]: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'June', 'Jul']
```



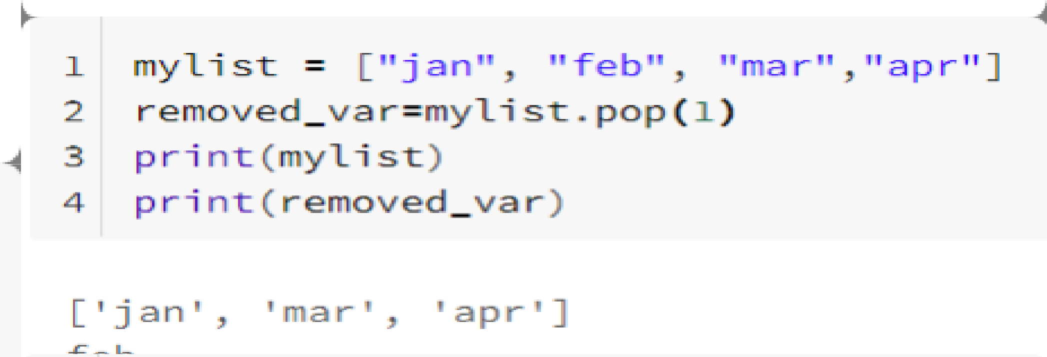
```
lenlist = [1,2,3,4,5,6,7,8,9,10]
len(lenlist)
```

```
Out[21]: 10
```



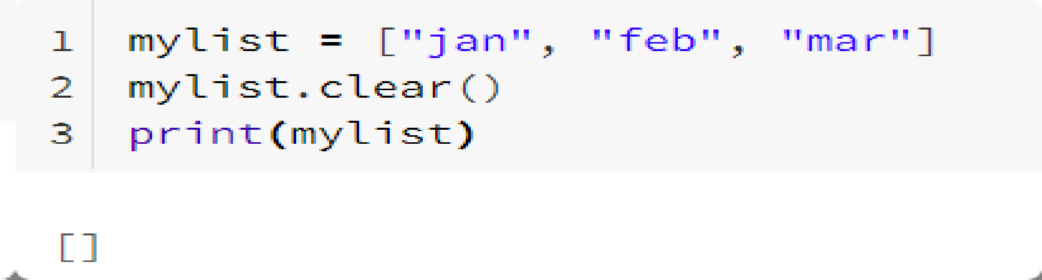
```
unsortlist = ['a','d','e','c','f','h','g','b']
unsortlist.sort(reverse=True)
unsortlist
```

```
Out[22]: ['h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']
```



```
mylist = ["jan", "feb", "mar","apr"]
removed_var=mylist.pop(1)
print(mylist)
print(removed_var)
```

```
['jan', 'mar', 'apr']
feb
```



```
mylist = ["jan", "feb", "mar"]
mylist.clear()
print(mylist)
```

```
[]
```

Tuple

A tuple is a collection which is ordered and unchangeable. In Python tuples are written with round brackets ().

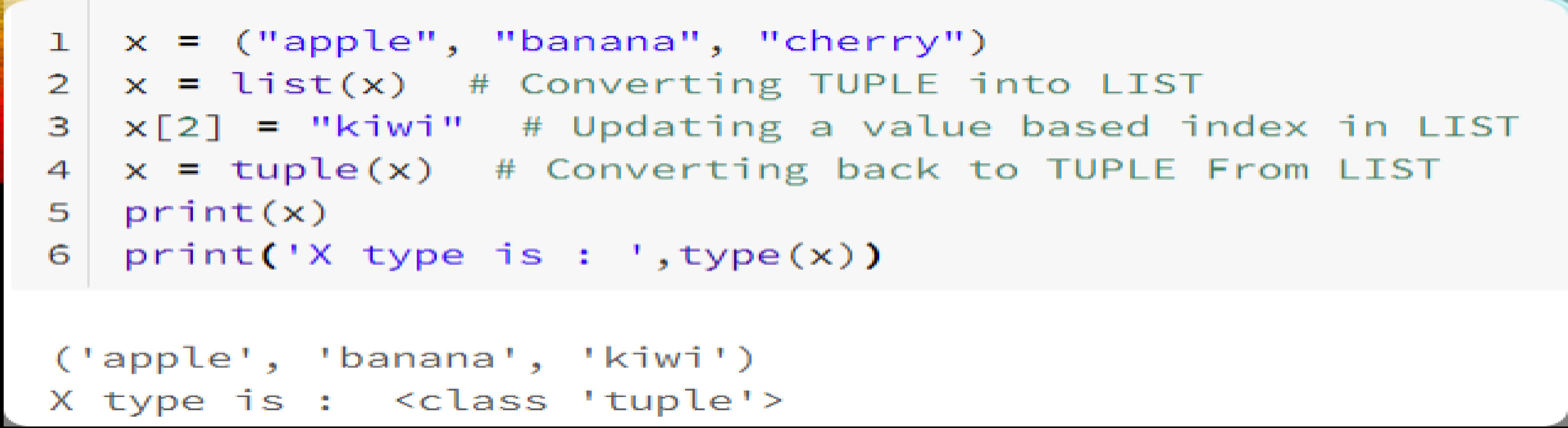
- ➢ Python tuple is much like a list except that it is immutable or unchangeable once created.
- ➢ Tuples use parentheses and creating them is as easy as putting different items separated by a comma between parentheses.

Slicing :

- ➢ If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

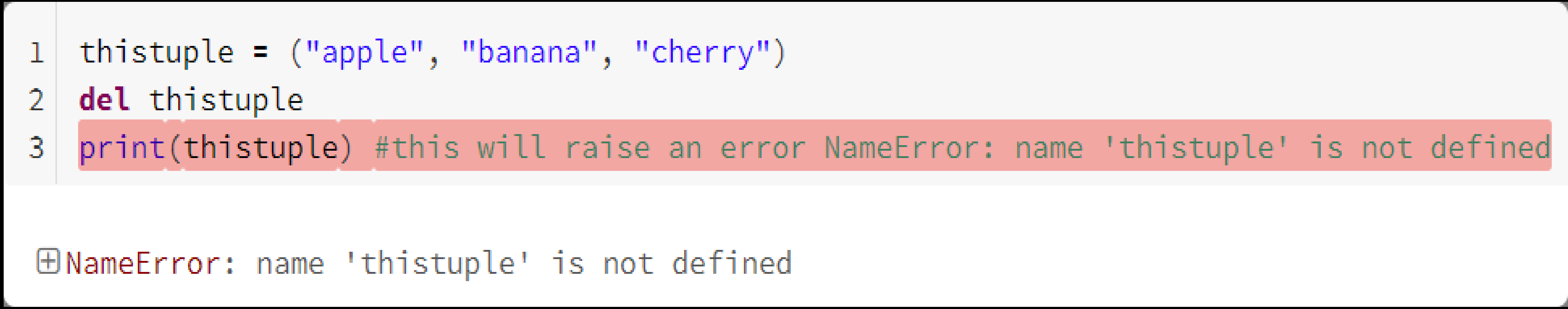
Tuple Values :

- Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called. But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple



```python
x = ("apple", "banana", "cherry")
x = list(x)   # Converting TUPLE into LIST
x[2] = "kiwi"   # Updating a value based index in LIST
x = tuple(x)   # Converting back to TUPLE From LIST
print(x)
print('X type is : ',type(x))
```

```
('apple', 'banana', 'kiwi')
X type is :  <class 'tuple'>
```



```python
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error NameError: name 'thistuple' is not defined
```

```
⊞NameError: name 'thistuple' is not defined
```

Dictionaries

A dictionary is like a list, but more general. In a list, the index positions have to be integers; in a dictionary, the indices can be (almost) any type. The function dict creates a new dictionary with no items. Because dict is the name of a built-in function, you should avoid using it as a variable name. A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

- dct = {"Name": "John",
  "Age": 25,
  "Height": 6.1}

- print(dct["Name"])
- print(dct["Age"])
- print(dct["Height"])

INPUT FUNCTION

- number1 = input("Enter first number: ")
- number2 = input("Enter second number: ")
- sum = number1 + number2
- print("Result: ", sum)
- number1 = input("Enter first number: ")
- number2 = input("Enter second number: ")
- number1 = int(number1)
- number2 = int(number2)
- sum = number1 + number2
- print("Result: ", sum)

PRINT()

- Description:
- Displays output to the console

- Example:
- print('Hello, World!')

INPUT()

- Description:
- Takes user input

- Example:
- name = input('Enter your name: ')

LEN()

- Description:
- Returns the length of an object

- Example:
- len([1, 2, 3]) # Output: 3

TYPE()

- Description:
- Returns the type of an object

- Example:
- type(42) # Output: <class 'int'>

INT()

- Description:
- Converts a value to an integer

- Example:
- int('42') # Output: 42

FLOAT()

- Description:
- Converts a value to a float

- Example:
- float('3.14') # Output: 3.14

STR()

- Description:
- Converts a value to a string

- Example:
- str(42) # Output: '42'

RANGE()

- Description:
- Generates a sequence of numbers

- Example:
- list(range(5)) # Output: [0, 1, 2, 3, 4]

LIST()

- Description:
- Creates a list

- Example:
- list('abc') # Output: ['a', 'b', 'c']

DICT()

- Description:
- Creates a dictionary

- Example:
- dict(key='value') # Output: {'key': 'value'}

SET()

- Description:
- Creates a set

- Example:
- set([1, 2, 2, 3]) # Output: {1, 2, 3}

HELP()

- Description:
- Displays the documentation of a function

- Example:
- help(len)

SORTED()

- Description:
- Returns a sorted list

- Example:
- sorted([3, 1, 2]) # Output: [1, 2, 3]

SUM()

- Description:
- Returns the sum of elements

- Example:
- sum([1, 2, 3]) # Output: 6

MAX()

- Description:
- Returns the maximum element

- Example:
- max([1, 2, 3]) # Output: 3

MIN()

- Description:
- Returns the minimum element

- Example:
- min([1, 2, 3]) # Output: 1

ABS()

- Description:
- Returns the absolute value

- Example:
- abs(-5) # Output: 5

OPEN()

- Description:
- Opens a file

- Example:
- open('file.txt', 'r')

APPEND()

- Description:
- Adds an item to a list

- Example:
- lst = [1, 2]; lst.append(3) # Output: [1, 2, 3]

POP()

- Description:
- Removes and returns an item from a list

- Example:
- lst = [1, 2, 3]; lst.pop() # Output: 3

DEFINING FUNCTIONS

- A **function** is a reusable block of code designed to perform a specific computation or task. It consists of a name, a set of parameters (optional), and a sequence of statements that define its behavior. Here’s a breakdown of key points:

Defining a Function:

- Functions are defined using the def keyword.
- The syntax includes the function name followed by a parenthesized list of parameters (if any).
- The body of the function begins on the next line and must be indented.

Key Characteristics of Functions:

- A function is executed only when explicitly called.
- • It can accept inputs, called **parameters**, to customize its behavior.
- • A function can return a result to the caller using the return statement.

```
# Function definition
def greet(name):
    """This function greets the person with the provided name."""
    print(f"Hello, {name}!")

# Calling the function
greet("Alice")  # Output: Hello, Alice!
```

- def hello():
- print("Hello")

If we want to make our functions more dynamic, we can define parameters. These parameters can then be processed in the function code.

- def print_sum(number1, number2):
- print(number1 + number2)

RETURN Statement

- def add(number1, number2):
- return number1 + number2

Here we return the sum of the two parameters instead of printing it. But we can then use this result in our code.

- number3 = add(10, 20)
- print(add(10, 20))

Arguments (Parameters) in Functions

- Functions become more versatile and powerful when they accept **arguments**. Arguments allow you to pass information into a function, enabling it to work with different data during each call.

Types of Function Arguments:

1. **Positional Arguments**:
   - Passed to the function in the order they appear.
   - The function processes these arguments based on their position.

- 2. **Keyword Arguments**:
   - Passed using a key-value pair format (key=value).
   - The order does not matter as the arguments are identified by their names.

Key-Points:

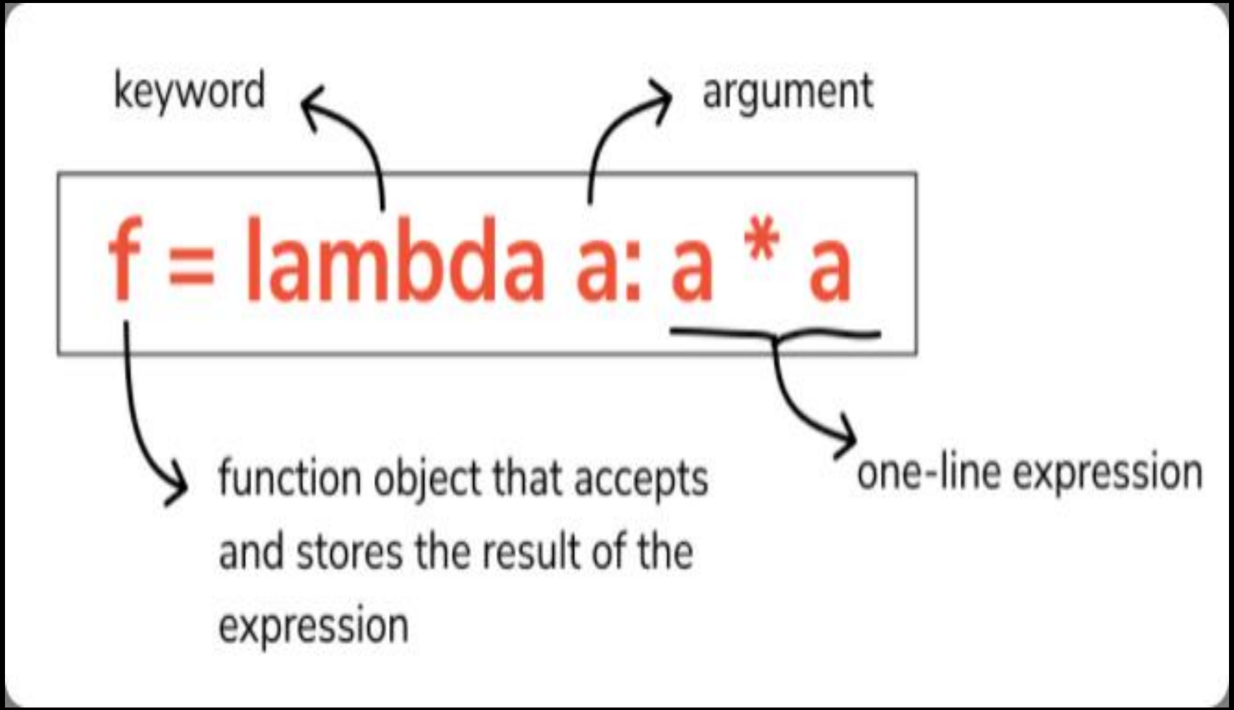
- **Passing Arguments**: Functions accept arguments to process new values and perform specific tasks.
- **Positional vs Keyword**:
- **Positional Arguments**: Depend on the order in which they are provided during the function call.
- **Keyword Arguments**: Explicitly associate values with parameter names, improving readability and reducing the chance of errors.

Example :

```
# Function with positional and keyword arguments
def calculate_area(length, width=5):
    """Calculate area of a rectangle."""
    return length * width

# Using positional arguments
print(calculate_area(10, 20))  # Output: 200

# Using a mix of positional and keyword arguments
print(calculate_area(length=10, width=15))  # Output: 150

# Using a single positional argument with default keyword value
print(calculate_area(10))  # Output: 50
```

Lambda Functions in Python

A **lambda function** is a small, anonymous function defined using the lambda keyword. It can take any number of arguments but is restricted to a single expression. The result of the expression is automatically returned.



```
f = lambda a: a * a
```

function object that accepts and stores the result of the expression

lambda arguments: expression

Characteristics:

1. **Anonymous**: Lambda functions do not require a name.
2. **Single Expression**: The function body can only have one line.
3. **Short and Concise**: Useful for simple operations.

Examples :

```
1# Define a lambda function to add two numbers
add = lambda x, y: x + y
print(add(5, 10))
 # Output: 15
2# Lambda function to calculate the square of a number
square = lambda x: x ** 2
print(square(4))
# Output: 16
# Use filter to extract even numbers from a list
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4, 6]

# Double each number in a list using map
numbers = [1, 2, 3, 4]
doubled = list(map(lambda x: x * 2, numbers))
print(doubled)  # Output: [2, 4, 6, 8]

# Sort a list of tuples by the second element
pairs = [(1, 'one'), (2, 'two'), (3, 'three')]
sorted_pairs = sorted(pairs, key=lambda x: x[1])
print(sorted_pairs)  # Output: [(1, 'one'), (3, 'three'), (2, 'two')]
```

Reading and Writing Files (file I/O)

- open() returns a file object, and is most commonly used with two arguments: open(filename, mode).
- The key function for working with files in Python is the open() function.
- The open() function takes two parameters; filename, and mode.
- There are four different methods (modes) for opening a file
- ➢ "r" - Read - Default value. Opens a file for reading, error if the file does not exist
- ➢ "a" - Append - Opens a file for appending, creates the file if it does not exist
- ➢ "w" - Write - Opens a file for writing, creates the file if it does not exist
- ➢ "x" - Create - Creates the specified file, returns an error if the file exists

f.readline() reads a single line from the file; a newline character (\n) is left at the end of the string, and is only omitted on the last line of the file if the file doesn’t end in a newline. This makes the return value unambiguous; if f.readline() returns an empty string, the end of the file has been reached, while a blank line is represented by '\n', a string containing only a single newline.

```
# Create a sample text file
with open("sample.txt", "w") as file:
    file.write("Hello, World!\n")
    file.write("\n")  # Blank line
    file.write("Welcome to Python.\n")

# Reading the file line by line using f.readline()
with open("sample.txt", "r") as file:
    while True:
        line = file.readline()
        # Check if end of file is reached
        if line == '':
            print("End of file reached.")
            break
        elif line == '\n':  # Check for blank line
            print("This is a blank line.")
        else:
            print(f"Read line: {line.strip()}")
```

- Reading and Writing Files
- `open()` returns a file object, and is most commonly used with two arguments: `open(filename, mode)`.

Write to an Existing File

- **Common Modes:**
- • **“r” (Read)**: Default mode; opens the file for reading.
- • **“w” (Write)**: Opens the file for writing. Overwrites existing content or creates a new file if it doesn’t exist.
- • **“a” (Append)**: Opens the file for appending new content to the end without modifying existing content.

```
  1# Writing new content to a file (overwriting if it exists)
  with open("example_write.txt", "w") as file:
   file.write("This is the first line of the file.\n")
   file.write("This will overwrite any existing content.\n")
print("File written using 'w' mode.")
```

```
# Appending content to an existing file
with open("example_write.txt", "a") as file:
    file.write("This line is added to the existing content.\n")
    file.write("Appending doesn't overwrite the file.\n")
print("File updated using 'a' mode.")
```