## ➢ Python define:

Python def keyword is **used to define a function**, it is placed before a function name that is provided by the user to create a user-defined function. In python, a function is a logical unit of code containing a sequence of statements indented under a name given using the “def” keyword.

**SYNTAX:**

```
>>> print("Hello, World!")
    Hello, World!
```

## ➢ Python Functions:

A function is a block of code that performs a specific task.

Suppose, you need to create a program to create a circle and color it. You can create two functions to solve this problem:

- create a circle function
- create a color function

Dividing a complex problem into smaller chunks makes our program easy to understand and reuse.

## ➢ Types of function:

There are two types of function in Python programming:

- **Standard library functions** - These are built-in functions in Python that are available to use.

- **User-defined functions** - We can create our own functions based on our requirements.

## ➢ Python Function Declaration example:

```
def function_name(arguments):
    # function body
return
```

Here,

- `def` - keyword used to declare a function
- `function_name` - any name given to the function
- `arguments` - any value passed to function
- `return` (optional) - returns value from a function

Let's see an example,

- **def greet():**
- **print('Hello World!')**

## ➢ Calling a Function in Python

Here's how we can call the `greet()` function in Python

```
# call the function
greet()
```

## Example: Python Function

```
def greet():
    print('Hello World!')
```

```
# call the function
greet()

print('Outside function')
```

**Output:**

```
Hello World!
Outside function
```

## Python Function Arguments

As mentioned earlier, a function can also have arguments. An argument is a value that is accepted by a function. For example,

```
# function with two arguments
def add_numbers(num1, num2):
    sum = num1 + num2
    print('Sum: ',sum)

# function with no argument
def add_numbers():
    # code
```

If we create a function with arguments, we need to pass the corresponding values while calling them. For example,

```
# function call with two values
add_numbers(5, 4)

# function call with no value
add_numbers()
```

Here, `add_numbers(5, 4)` specifies that arguments `num1` and `num2` will get values **5** and **4** respectively.

## The return Statement in Python

A Python function may or may not return a value. If we want our function to return some value to a function call, we use the `return` statement. For example,

```
def add_numbers():
    ...
    return sum
```

Here, we are returning the variable `sum` to the function call.

### Example 3: Add Two Numbers

```
# function that adds two numbers
def add_numbers(num1, num2):
    sum = num1 + num2
    return sum

# calling function with two values
result = add_numbers(5, 4)

print('Sum: ', result)

# Output: Sum: 9
```

## Python Library Functions

In Python, standard library functions are the built-in functions that can be used directly in our program. For example,

- `print()` - prints the string inside the quotation marks
- `sqrt()` - returns the square root of a number
- `pow()` - returns the power of a number

These library functions are defined inside the module. And, to use them we must include the module inside our program.

For example, `sqrt()` is defined inside the `math` module.

## Example 4: Python Library Function

```
import math

# sqrt computes the square root
square_root = math.sqrt(4)

print("Square Root of 4 is",square_root)

# pow() comptes the power
power = pow(2, 3)

print("2 to the power 3 is",power)
```

Run Code

**Output**

```
Square Root of 4 is 2.0
2 to the power 3 is 8
```

In the above example, we have used

- `math.sqrt(4)` - to compute the square root of **4**
- `pow(2, 3)` - computes the power of a number i.e. $\mathbf{2^3}$

### Variables in Python:

A Python variable is **a symbolic name that is a reference or pointer to an object**. Once an object is assigned to a variable, you can refer to the object by that.

## Variable Types in Python

In many programming languages, variables are statically typed. That means a variable is initially declared to have a specific data type, and any value assigned to

it during its lifetime must always have that type.Variables in Python are not subject to this restriction. In Python, a variable may be assigned a value of one type and then later re-assigned a value of a different type:

```
>>> var = 23.5
>>> print(var)
23.5

>>> var = "Now I'm a string"
>>> print(var)
Now I'm a string
```

## Variable Names:

The examples you have seen so far have used short, terse variable names like `m` and `n`. But variable names can be more verbose. In fact, it is usually beneficial if they are because it makes the purpose of the variable more evident at first glance.

Officially, variable names in Python can be any length and can consist of uppercase and lowercase letters (`A-Z`, `a-z`), digits (`0-9`), and the underscore character (_). An additional restriction is that, although a variable name can contain digits, the first character of a variable name cannot be a digit.

## EXAMPLE:

```
>>> name = "Bob"
>>> Age = 54
>>> has_W2 = True
>>> print(name, Age, has_W2)
Bob 54 True
```

## But this one is not, because a variable name can’t begin with a digit:

```
>>> 1099_filed = False
SyntaxError: invalid token
```

## Reserved Words (Keywords)

There is one more restriction on identifier names. The Python language reserves a small set of keywords that designate special language functionality. No object can have the same name as a reserved word.

**In Python 3.6, there are 33 reserved keywords**:

| **Python Keywords** | | | |
|---|---|---|---|
| False | def | if | raise |
| None | del | import | return |
| True | elif | in | try |
| and | else | is | while |
| as | except | lambda | with |
| assert | finally | nonlocal | yield |
| break | for | not | |
| class | from | or | |
| continue | global | pass | |

**EXAMPLE:**

```
        >>> for = 3
SyntaxError: invalid syntax
```

# Python Data Types

Built-in Data Types: In programming, data type is an important concept.

Variables can store data of different types, and different types can do different things.

Python has the following data types built-in by default, in these categories:

| | |
|---|---|
| Text Type: | `str` |
| Numeric Types: | `int`, `float`, `complex` |
| Sequence Types: | `list`, `tuple`, `range` |
| Mapping Type: | `dict` |

| | |
|---|---|
| Set Types: | `set`, `frozenset` |
| Boolean Type: | `bool` |
| Binary Types: | `bytes`, `bytearray`, `memoryview` |
| None Type: | `NoneType` |

# Getting the Data Type:

You can get the data type of any object by using the `type()` function:

## EXAMPLE

Print the data type of the variable x:

```
x = 5
print(type(x))
```

| | |
|---|---|
| x = "Hello World" | str |
| x = 20 | int |
| x = 20.5 | float |
| x = 1j | complex |
| x = ["apple", "banana", "cherry"] | list |
| x = ("apple", "banana", "cherry") | tuple |
| x = range(6) | range |
| x = {"name" : "John", "age" : 36} | dict |
| x = True | bool |
| x = b"Hello" | bytes |
| x = bytearray(5) | bytearray |
| x = memoryview(bytes(5)) | memoryview |
| x = None | NoneType |

# Python Lists

```
mylist = ["apple", "banana", "cherry"]
```

# List:

Lists are used to store multiple items in a single variable.

Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.

Lists are created using square brackets:

**EXAMPLE:**

Create a List:

```
mylist = ["apple", "banana", "cherry"]
print(mylist)
```

# List Items

List items are ordered, changeable, and allow duplicate values.

List items are indexed, the first item has index `[0]`, the second item has index `[1]` etc.

# Ordered

When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

# Allow Duplicates

Since lists are indexed, lists can have items with the same value:

## EXAMPLE

Lists allow duplicate values:

```
thislist = ["apple", "banana", "cherry", "apple", "cherry"]
print(thislist)
```

# List Length

To determine how many items a list has, use the `len()` function:

**EXAMPLE:**

```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))
```

**OUTPUT:**

**3**

# List Items - Data Types

## Example

String, int and boolean data types:

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
```

# type()

From Python's perspective, lists are defined as objects with the data type 'list':

```
<class 'list'>
```

## Example

What is the data type of a list?

```
mylist = ["apple", "banana", "cherry"]
print(type(mylist))
```

**OUTPUT:**

**<class 'list'>**

# Python Collections (Arrays):

There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **set** is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered** and changeable. No duplicate members.

# • Python Operators

Operators are used to perform operations on variables and values.

In the example below, we use the + operator to add together two values.

```
print(10 + 5)
```

**output : 15**

**Python divides the operators in the following groups:**

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

# Tuple:

Tuples are used to store multiple items in a single variable.

Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.

A tuple is a collection which is ordered and **unchangeable**.

Tuples are written with round brackets.

```
mytuple = ("apple", "banana", "cherry")
```

## Example

Create a Tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

# Python Sets:

```
myset = {"apple", "banana", "cherry"}
```

## Set

Sets are used to store multiple items in a single variable.

Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.

A set is a collection which is *unordered*, *unchangeable**, and *unindexed*.

## Example

Create a Set:

```
myset = {"apple", "banana", "cherry"}
print(myset)
```

# Unordered

Unordered means that the items in a set do not have a defined order.

Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

# Unchangeable

Set items are unchangeable, meaning that we cannot change the items after the set has been created.

# Duplicates Not Allowed

Sets cannot have two items with the same value.

## Example

Duplicate values will be ignored:

```
myset = {"apple", "banana", "cherry", "apple"}

print(myset)
```

**OUTPUT:**

**{'banana', 'cherry', 'apple'}**

# Python Dictionaries

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered*, changeable and do not allow duplicates.

### Example

Create and print a dictionary:

```
newdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(newdict)
```

# Dictionary Items

Dictionary items are ordered, changeable, and does not allow duplicates.

Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

# Python If ... Else

## Python Conditions and If statements

Python supports the usual logical conditions from mathematics:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`

These conditions can be used in several ways, most commonly in "if statements" and loops.

An "if statement" is written by using the `if` keyword.

### Example:

```
a = 33
b = 33
```

```
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

**OUTPUT:**

**a and b are equal.**

# Python Loops

Python has two primitive loop commands:

- `while` loops
- `for` loops

# The while Loop

With the `while` loop we can execute a set of statements as long as a condition is true.

## Example:

Print i as long as i is less than 6:

```
i = 1
while i < 6:
print(i)
i += 1
```

**OUTPUT:**

**1**
**2**
**3**
**4**
**5**

# Python For Loops

A `for` loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the `for` keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the `for` loop we can execute a set of statements, once for each item in a list, tuple, set etc.

## **Example** Print each fruit in a fruit list:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

**OUTPUT:**

**apple**
**banana**
**cherry**

- The `for` loop does not require an indexing variable to set beforehand.