Function return

- Every function in Python returns something. If the function doesn't have any return statement, then it returns `None`.

```
def print_something(s):
    print('Printing...', s)
output = print_something('Hi')
print(f'A function without return statement returns {output}')
```

- The function ends when the `return` statement is reached

```
def print_something():
    return
    print("Cannot reach this line")
print_something()
```

- Python return statement with expression

```
def add(x, y):
    return x + y
output = add(5, 4)
print(f'Output of add(5, 4) function is {output}')
```

Useful Python functions

- **enumerate()** returns the length of an iterable and loops through its items simultaneously. Thus, while printing each item in an iterable data type, it simultaneously outputs its index.

```
fruits = ["grape", "apple", "mango"]
for i, j in enumerate(fruits):
      print(i, j)

#Output:
#0 grape
#1 apple
#2 mango
```

Useful Python functions

- **eval()** lets you perform mathematical operations on integers or floats, even in their string forms. It's often helpful if a mathematical calculation is in a string format.

```
g = "(4 * 5)/4"
d = eval(g)

print(d)  # 5.0
```

Useful Python functions

- **type()** function is mostly used for debugging purposes. This function returns the type of the object referenced by a variable name

```
lst = []
type(lst) # output: <class 'list'>
```

- **isinstance()** function returns `True` if the specified object is of the specified type, otherwise `False`.

```
x = 5
isinstance("Hello", (float, int, str, list, dict, tuple))  # True
```

Useful Python operator

- `is` check for conformity (`True` or `False`)

```
x = None
print(x is None) # True
```

- `is not` check for conformity (`True` or `False`)

```
x = None
print(x is not None) # False
```

- `in` is used to check if the variable is in the list (`True` or `False`)

```
lst= [1,2,3,4,5,6]
print(3 in lst) # True
print(0 in lst) # False
```

Modules (hour 8)

- Introduction to modules.
- Importing modules and using their functions.
- Creating and using your own modules.

Introduction to *modules*

In Python, a module is a file containing Python code that defines functions, classes, and variables that can be used in other Python programs. Modules provide a way to organize code and promote code reusability. By importing modules, you can access their defined functionality and use it in your own programs.

Creating Modules

You can create your own modules by writing Python code in a .py file. Each file represents a module, and you can define functions, classes, and variables within the file. For example, you can create a module named my_module.py containing a function:

```
# my_module.py
def greet(name):
    print(f"Hello, {name}!")
```

Importing Modules

To use the functionality defined in a module, you need to import it into your Python program. Python provides several ways to import modules:

```
import my_module

my_module.greet("Alice")  # Output: "Hello, Alice!“
```

- Importing Specific Items

```
from my_module import greet

greet("Bob")  # Output: "Hello, Bob!“
```

- Module Aliasing

```
import my_module as mm

mm.greet("Charlie")  # Output: "Hello, Charlie!"
```

Creating module package

To create a module consisting of multiple files in a single folder, Python needs to place the file (can be empty) `__init__.py` in the folder used for the module name (folder name will be the name of the module).

Inside `__init__.py` you can precise importing rules:

```
from .file_a import method_a
from .file_b import method_b
```

Standard package manager for Python

pip is the package installer for Python. Use **pip** in a secure manner to install a Python application and its dependencies during deployment.

```
pip install SomeProject1, SomeProject2
python3 -m pip install SomeProject1, SomeProject2
```

- `==` To install a specific version

```
pip install SomeProject==1.4
```

- `~=` To install a version that's compatible with a certain version
- `>=` To install a specific version or newer
- `>=1,<2` To install greater than or equal to one version and less than another

Storing of the list of installed packages

- You can use the pip package manager itself to save the list of installed packages to a text file. Open your command prompt or terminal and run the following command:

```
pip freeze > requirements.txt
```

- To install the packages from the file (for example, from requirements.txt):

```
pip install -r requirements.txt
```

Virtual environment in Python

Using a virtual environment allows you to manage different sets of packages and their versions for different projects. It helps keep project dependencies isolated and avoids conflicts between packages used in different projects.

- **Create** a new virtual environment:
  - Open your terminal or command prompt.
  - Navigate to the directory where you want to create the virtual environment.
  - Run the following command to create a new virtual environment (replace myenv with the desired name for your environment):

`python3 -m venv myenv` or `python3 -m venv /path/to/myenv`

- **Activate** the virtual environment:
  - On Windows: `myenv\Scripts\activate.bat`
  - On macOS and Linux: `source myenv/bin/activate`

After activation, your terminal prompt should change to indicate that you are now working within the virtual environment.

- **Deactivate** the virtual environment: To exit the virtual environment and return to your system's global Python environment, use the following command:

`deactivate`

Exceptions in Python (*hour 9*)

- Exception handling.
- Handling errors with try-except blocks.
- Raising and catching specific exceptions.

Exception handling in Python

Exception handling in Python allows you to catch and handle errors or exceptions that may occur during the execution of your program. It helps you handle these exceptions gracefully and prevents your program from crashing. Python provides a try-except block for handling exceptions. Here's the basic syntax:

```
try:
    # Code that might raise an exception
except ExceptionType1:
    # Code to handle ExceptionType1
except ExceptionType2:
    # Code to handle ExceptionType2
else:
    # Code to execute if no exception is raised
finally:
    # Code that will always be executed, whether an exception occurred or not
```

Exception handling in Python

```
try:
    x = 10
    y = 0
    result = x / y
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Division by zero")
except TypeError:
    print("Error: Invalid operand types")
else:
    print("No exception occurred")
finally:
    print("Finally block executed")

# Output:
# Error: Division by zero
# Finally block executed
```

Raising exceptions

In Python, you can raise exceptions explicitly using the raise statement and catch specific exceptions using the try-except block. This allows you to handle specific error conditions in your code. Here's how you can raise and catch specific exceptions in Python:

```
raise ValueError("Invalid input")
```

Catching specific exceptions

- Catching a Specific Exception: To catch a specific exception, you can use the `try-except` block with one or more except clauses. Each `except` clause specifies the type of exception it can handle. If an exception of the specified type occurs within the try block, the corresponding except block is executed:

```
try:
    # Code that might raise an exception
except ValueError as e:
    # Code to handle ValueError
    print("ValueError occurred:", str(e))
except TypeError as e:
    # Code to handle TypeError
    print("TypeError occurred:", str(e))
```

- Catching Multiple Exceptions:

```
try:
    # Code that might raise an exception
    # ...
except (ValueError, TypeError) as e:
    # Code to handle ValueError or TypeError
    print("Exception occurred:", str(e))
```