PYTHON CƠ BẢN (Buổi 2)

AI Academy Vietnam

Content

II. Python Flow Control

1. if … else
2. for
3. while
4. break and continue
5. pass

II.1. `if..else` statement

- The **`if`**…**`elif`**…**`else`** statement is used in Python for decision making.
- Decision making is required when we want to execute a code only **if a certain condition is satisfied**.

1

```
if test expression:
        statement(s)
```

2

```
if test expression:
        Body of if
else:
        Body of else
```

3

```
if test expression:
        Body of if
elif test expression:
        Body of elif
else:
        Body of else
```

II.1. `if..else` statement

1

```
if test expression:
        statement(s)
```

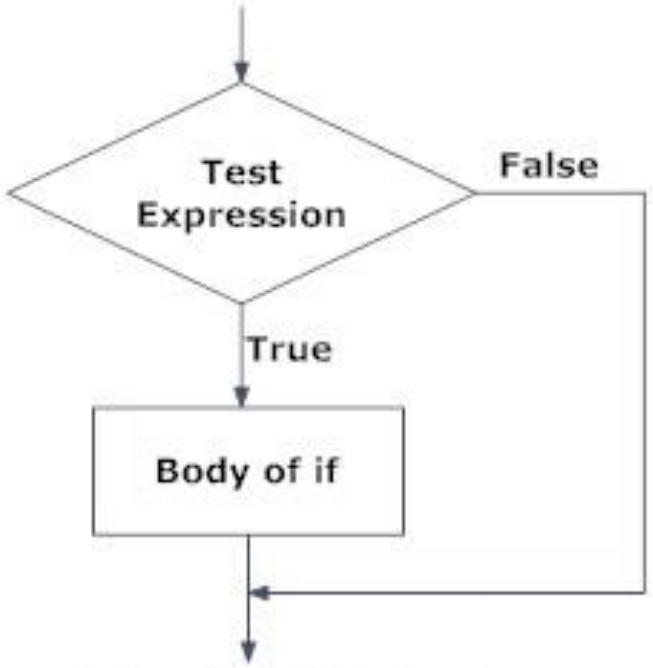


Fig: Operation of if statement

```
num = 3
if num > 0:
   print(num, "is a positive number.")
print("This is always printed.")
num = -1
if num > 0:
   print(num, "is a positive number.")
print("This is also always printed.")
```

```
3 is a positive number
This is always printed
This is also always printed.
```

II.1. `if..else` statement

2

```
if test expression:
        Body of if
else:
        Body of else
```

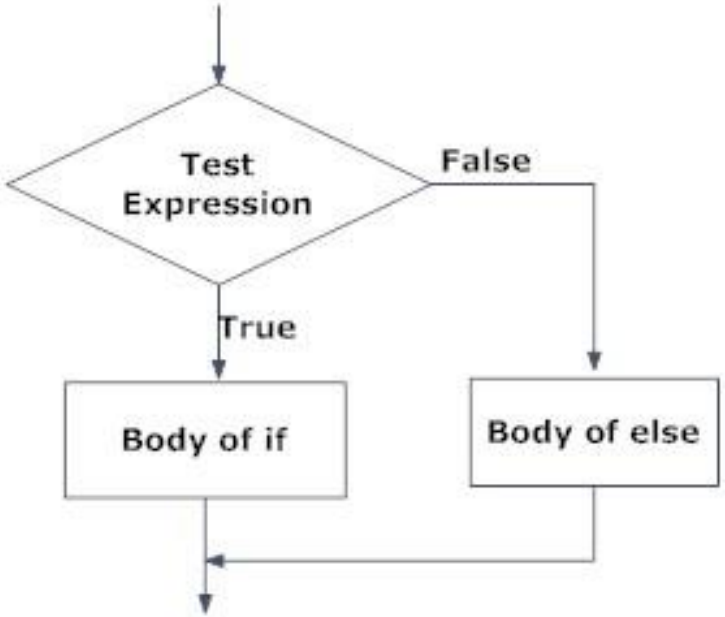


Fig: Operation of if...else statement

```
num = 3
# Try these two variations as well.
# num = -5
# num = 0
if num >= 0:
        print("Positive or Zero")
else:
        print("Negative number")
```

```
Positive or Zero
```

II.1. `if..else` statement

3

```
if test expression:
        Body of if
elif test expression:
        Body of elif
else:
        Body of else
```

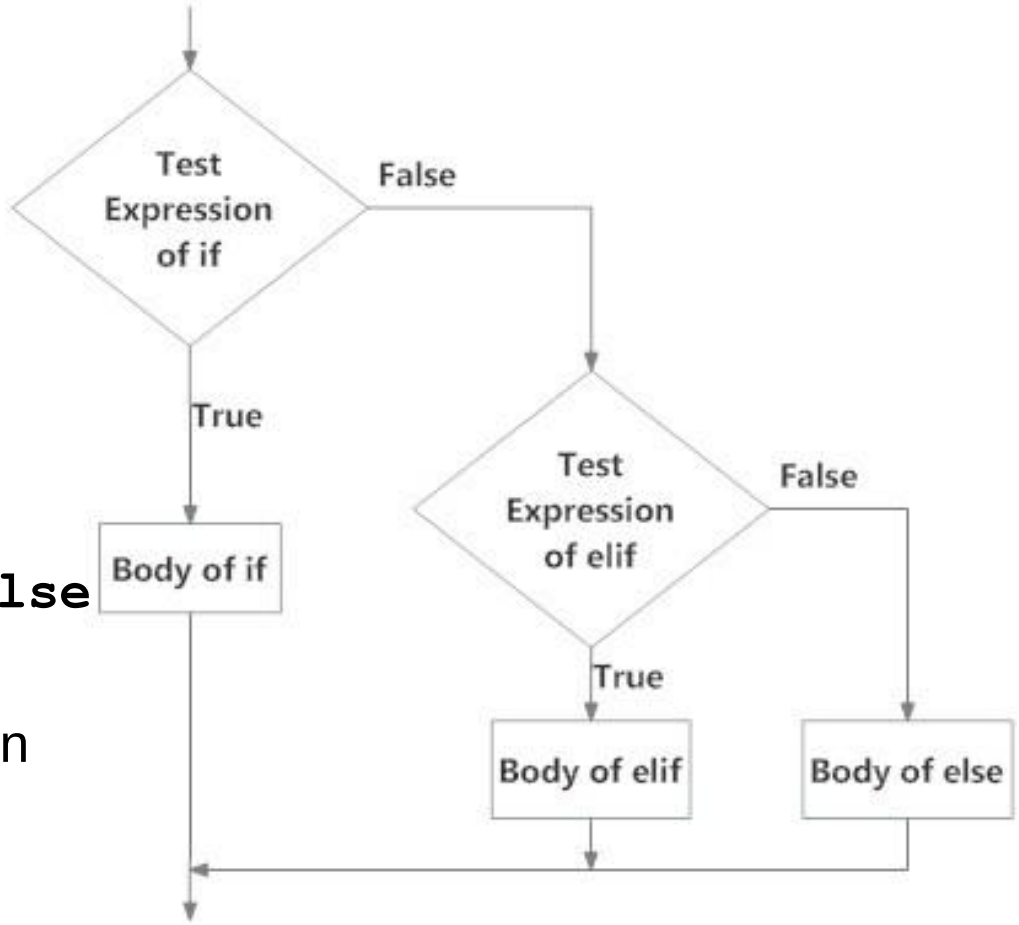


Fig: Operation of if...elif...else statement

- Only **one block** among the several `if...elif...else` blocks is executed according to the condition.
- The `if` block can have only one `else` block. But it can have multiple `elif` blocks.

II.1. `if..else` statement

```
num = 3.4
# Try these two variations as well:
# num = 0
# num = -4.5
if num > 0:
        print("Positive number")
elif num == 0:
        print("Zero")
else: print("Negative number")
```

```
if num >= 0:
        if num == 0:
                print("Zero")
        else:
                print("Positive number")
else:
        print("Negative number")
```

```
Positive number
```

II.2. `for` statement

- The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects.
- Iterating over a sequence is called traversal.

```
for val in sequence:
        Body of for
```

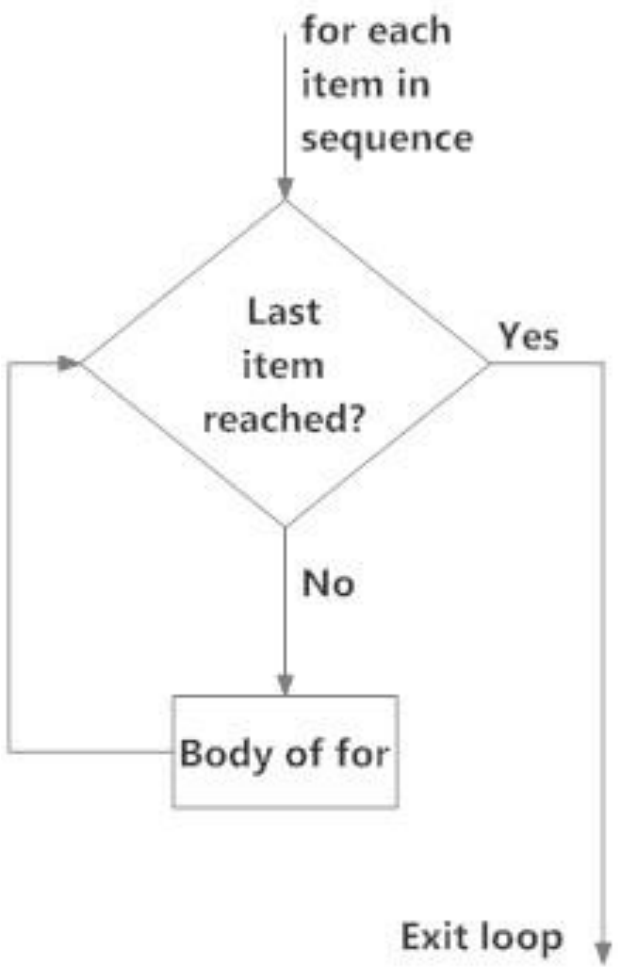


Fig: operation of for loop

II.2. `for` statement

```
# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
# variable to store the sum
sum = 0
# iterate over the list
for val in numbers:
        sum = sum+val
print("The sum is", sum)
```

```
The sum is 60
```

Only want to calculate the sum of odd numbers in the list?

```
# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
# variable to store the sum
sum = 0
# iterate over the list
for val in numbers:
        if val % 2 == 1:
                sum = sum+val
print("The sum is", sum)
```

```
The sum is 24
```

II.2. `for` statement

- The **`range()`** function: **`range(start, stop,step_size)`**

```
print(range(10))
print(list(range(10)))
print(list(range(2, 8)))
print(list(range(2, 20, 3)))
```

```
range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 3, 4, 5, 6, 7]
[2, 5, 8, 11, 14, 17]
```

II.2. `for` statement

```
# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
# variable to store the sum
sum = 0
# iterate over the list
for val in numbers:
        sum = sum+val
print("The sum is", sum)
```

```
# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
# variable to store the sum
sum = 0
# iterate over the list
for i in range(len(numbers)):
        sum = sum+numbers[i]
print("The sum is", sum)
```

II.2. `for` statement

- **`for`** loop with **`else:`**
  - The **`else`** part is executed if the items in the sequence used in **`for`** loop exhausts.
  - The **`break`** keyword can be used to stop a **`for`** loop. In such cases, the **`else`** part is ignored.

```
digits = [0, 1, 5]
for i in digits:
        print(i)
else:
        print("No items left.")
```

```
0
1
5
No items left.
```

II.2. `for` statement

```
# program to display student's marks from record
student_name = 'Soyuj'
marks = {'James': 90, 'Jules': 55, 'Arthur': 77}
for student in marks:
        if student == student_name:
                print(marks[student])
                break
else: print('No entry with that name found.')
```

```
No entry with that name found.
```

II.3. `while` statement

- The **`while`** loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.
- We generally use this loop when we don't know the number of times to iterate beforehand.

```
while test_expression:
        Body of while
```

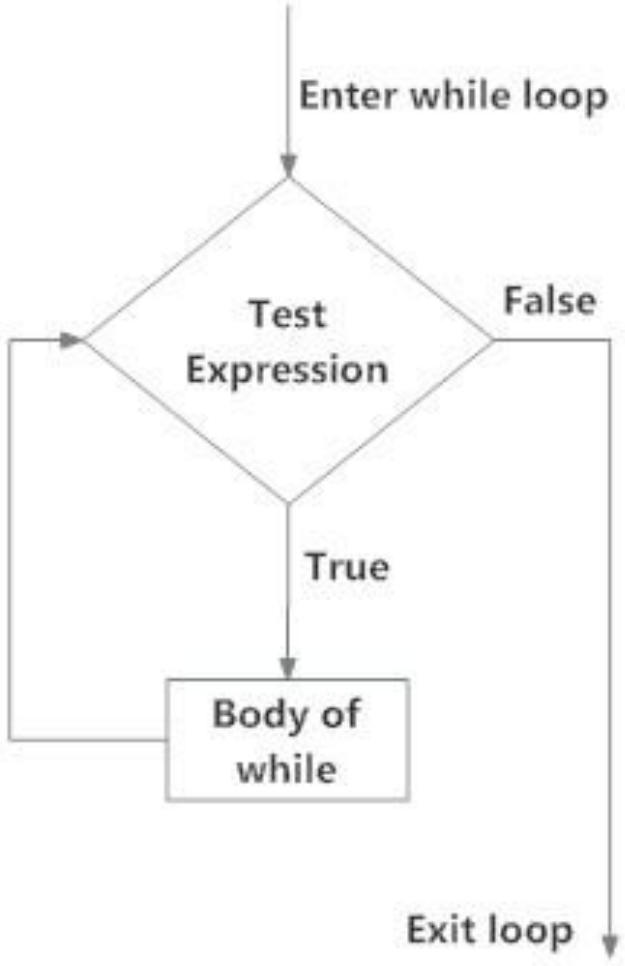


Fig: operation of while loop

II.3. `while` statement

```
n = 10
# initialize sum and counter
sum = 0
i = 1
while i <= n:
        sum = sum + i
        i = i+1 # update counter
# print the sum
print("The sum is", sum)
```

```
The sum is 55
```

II.3. `while` statement

```
counter = 0
while counter < 3:
        print("Inside loop")
        counter = counter + 1
else:
        print("Inside else")
```

II.4. **`break`** and **`continue`** statements

- **`break`** and **`continue`** statements can alter the flow of a normal loop.

```
for var in sequence:
    # codes inside for loop
    if  condition:
        break
    # codes inside for loop

# codes outside for loop
```

```
for var in sequence:
    # codes inside for loop
    if  condition:
        continue
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        break
    # codes inside while loop

# codes outside while loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        continue
    # codes  inside while loop

# codes outside while loop
```

II.4. **`break`** and **`continue`** statements

```
for val in "string":
        if val == "i":
                break
        print(val)
print("The end")
```

```
s
t
r
The end
```

```
for val in "string":
        if val == "i":
                continue
        print(val)
print("The end")
```

```
s
t
r
n
g
The end
```

II.5. `pass` statement

- The **`pass`** statement is a null statement.
- The difference between a comment and a **`pass`** statement in Python is that while the interpreter ignores a comment entirely, pass is not ignored.

```
sequence = {'p', 'a', 's', 's'}
for val in sequence:
        pass
```

```
def function(args):
        pass
```

```
class Example:
        pass
```

III. Python Functions

1. What is a function
2. Function Argument
3. Recursive Function
4. Anomymous Function
5. Global, Local and Nonlocal
6. Global Keyword
7. Modules
8. Package

III.1. What is a function?

- A function is a group of related statements that performs a specific task.
- Advantages of functions:
  - Help break our program into smaller and modular chunks.
  - Make programs more organized and manageable.
  - Avoids repetition and makes the code reusable.

```
def function_name(parameters):
        """docstring"""
        statement(s)
```

III.1. What is a function?

```
def functionName():
    ... .. ...
    ... .. ...

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

functionName();

... .. ...
... .. ...
```

```python
def greet(name):
        """ This function greets to
        the person passed in as
        a parameter """
        print("Hello, " + name + ". Good morning!")
greet('Paul')
print(greet("May"))
print(greet.__doc__)
```

```
Hello, Paul. Good morning!
Hello, May. Good morning!
None
This function greets to
the person passed in as
a parameter
```

III.1. What is a function?

- Scope and Lifetime of variables:
  - Parameters and variables defined inside a function have a local scope.
  - The variables inside a function exits in the memory as long as the function executes. They are destroyed once we return from the function.

```
def my_func():
        x = 10
        print("Value inside function:",x)
x = 20
my_func()
print("Value outside function:",x)
```

```
Value inside function: 10
Value outside function: 20
```

III.1. What is a function?

- Scope and Lifetime of variables:
  - Parameters and variables defined inside a function have a local scope.
  - The variables inside a function exits in the memory as long as the function executes. They are destroyed once we return from the function.

```
def my_func():
        x = 10
        print("Value inside function:",x)
x = 20
my_func()
print("Value outside function:",x)
```

```
Value inside function: 10
Value outside function: 20
```

III.2. Function Arguments

```
def greet(name, msg):
        """This function greets to the person with the provided message"""
        print("Hello", name + ', ' + msg)
greet("Monica", "Good morning!")
#greet("Monica")
```

```
Hello Monica, Good morning!
```

```
TypeError: greet() missing 1 required positional argument: 'msg'
```

III.2. Function Arguments

- Default Arguments:
  - It is optional during a call. If a value is provided, it will overwrite the default value.
  - Any number of arguments in a function can have a default value
  - All the arguments to its right must also have default values.

```
def greet(name, msg="Good morning!"):
        """ This function greets to the person with the provided message.
        If the message is not provided, it defaults to "Good morning!" """
        print("Hello", name + ', ' + msg)
greet("Kate")
greet("Bruce", "How do you do?")
```

```
Hello Kate, Good morning!
Hello Bruce, How do you do?
```

III.2. Function Arguments

- Keyword Arguments:
  - Python allows functions to be called using keyword arguments
  - The order (position) of the arguments can be changed.
  - keyword arguments must follow positional arguments.

```
def greet(name, msg
        """ This function greets to the person with the provided message.
        If the message is not provided, it defaults to "Good morning!" """
        print("Hello", name + ', ' + msg)
# 2 keyword arguments
greet(name = "Bruce",msg = "How do you do?")
# 2 keyword arguments (out of order)
greet(msg = "How do you do?",name = "Bruce")
#1 positional, 1 keyword argument
greet("Bruce", msg = "How do you do?")
```

III.3. Recursive Function

- A recursive function: a function that calls itself

```
def recurse():
    ...
    recurse()
    ...

recurse()
```

recursive call

```
def factorial(x):
        """This is a recursive function
        to find the factorial of an integer"""
        if x == 1:
                return 1
        else:
                return (x * factorial(x-1))
num = 3
print("The factorial of", num, "is", factorial(num))
```

```
The factorial of 3 is 6
```

III.3. Recursive Function

```
def factorial(x):
        """This is a recursive function
        to find the factorial of an integer"""
        print("Calculate factorial of", x)
        if x == 1:
                return 1
        else:
                return (x * factorial(x-1))
num = 3
print("The factorial of", num, "is", factorial(num))
```

```
Calculate factorial of 3
Calculate factorial of 2
Calculate factorial of 1
The factorial of 3 is 6
```

III.3. Recursive Function

```
x = factorial(3)
```

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

3 2

3*2 = 6

is returned

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

2 1

2*1 = 2

is returned

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

1

is returned

III.3. Recursive Function

- Our recursion ends when the base condition is satisfied
- Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.
- The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.
- By default, the maximum depth of recursion is 1000.

III.3. Recursive Function

- Advantages of Recursion
  - Recursive functions make the code look clean and elegant.
  - A complex task can be broken down into simpler sub-problems using recursion.
  - Sequence generation is easier with recursion than using some nested iteration.
- Disadvantages of Recursion
  - Sometimes the logic behind recursion is hard to follow through.
  - Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
  - Recursive functions are hard to debug.

III.4. Anonymous/Lambda Function

- An anonymous function is a function that is defined without a name.
- Anonymous functions are defined using the **`lambda`** keyword.

```
lambda arguments: expression
```

```
# Program to show the use of lambda functions
double = lambda x: x * 2
print(double(5))
```

III.4. Anonymous/Lambda Function

- Example use with filter(): The filter() function in Python takes in a function and a list as arguments.

```
# Program to filter out only the even items from a list
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(filter(lambda x: (x%2 == 0) , my_list))
print(new_list)
```

```
[4, 6, 8, 12]
```

III.4. Anonymous/Lambda Function

- Example use with map(): The map() function in Python takes in a function and a list as arguments.

```
# Program to filter out only the even items from a list
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(map(lambda x: x*2, my_list))
print(new_list)
```

```
[2, 10, 8, 12, 16, 22, 6, 24]
```

III.5. Global, Local and Nonlocal variables

- A global variable: a variable declared outside of the function or in global scope. This means that a global variable can be accessed inside or outside of the function.
- A local variable: a variable declared inside the function's body or in the local scope.

```
x = "global "
def foo():
        global x
        y = "local"
        x = x * 2
        print(x)
        print(y)
foo()
```

```
x = 5
def foo():
        x = 10
        print("local x:", x)
foo()
print("global x:", x)
```

III.5. Global, Local and Nonlocal variables

- Nonlocal Variables: are used in nested functions whose local scope is not defined. This means that the variable can be neither in the local nor the global scope.

```
def outer():
        x = "local"
        def inner():
                nonlocal x
                x = "nonlocal"
                print("inner:", x)
        inner()
        print("outer:", x)
outer()
```

```
inner: nonlocal
outer: nonlocal
```

III.6. Global Keyword

- `global` keyword allows you to modify the variable outside of the current scope.
- It is used to create a global variable and make changes to the variable in a local context.
- The basic rules for global keyword in Python are:
  - When we create a variable inside a function, it is local by default.
  - When we define a variable outside of a function, it is global by default. You don't have to use `global` keyword.
  - We use `global` keyword to read and write a global variable inside a function.
  - Use of `global` keyword outside a function has no effect.

III.6. Global Keyword

```
def foo():
        x = 20
        def bar():
                global x
                x = 25
        print("Before calling bar: ", x)
        print("Calling bar now")
        bar()
        print("After calling bar: ", x)
foo()
print("x in main: ", x)
```

```
Before calling bar: 20
Calling bar now
After calling bar: 20
x in main: 25
```

III.7. Modules

- Modules refer to a file containing Python statements and definitions.
- We use modules to break down large programs into small manageable and organized files.
- Modules provide reusability of code.
- We can define our most used functions in a module and import it
- While importing a module, Python looks at several places. Interpreter first looks for a built-in module. Then(if built-in module not found), Python looks into a list of directories defined in sys.path. The search is in this order.
  - The current directory.
  - PYTHONPATH (an environment variable with a list of directories).
  - The installation-dependent default directory.
- https://docs.python.org/3/py-modindex.html

III.7. Modules

- Type the our functions and save it as example.py

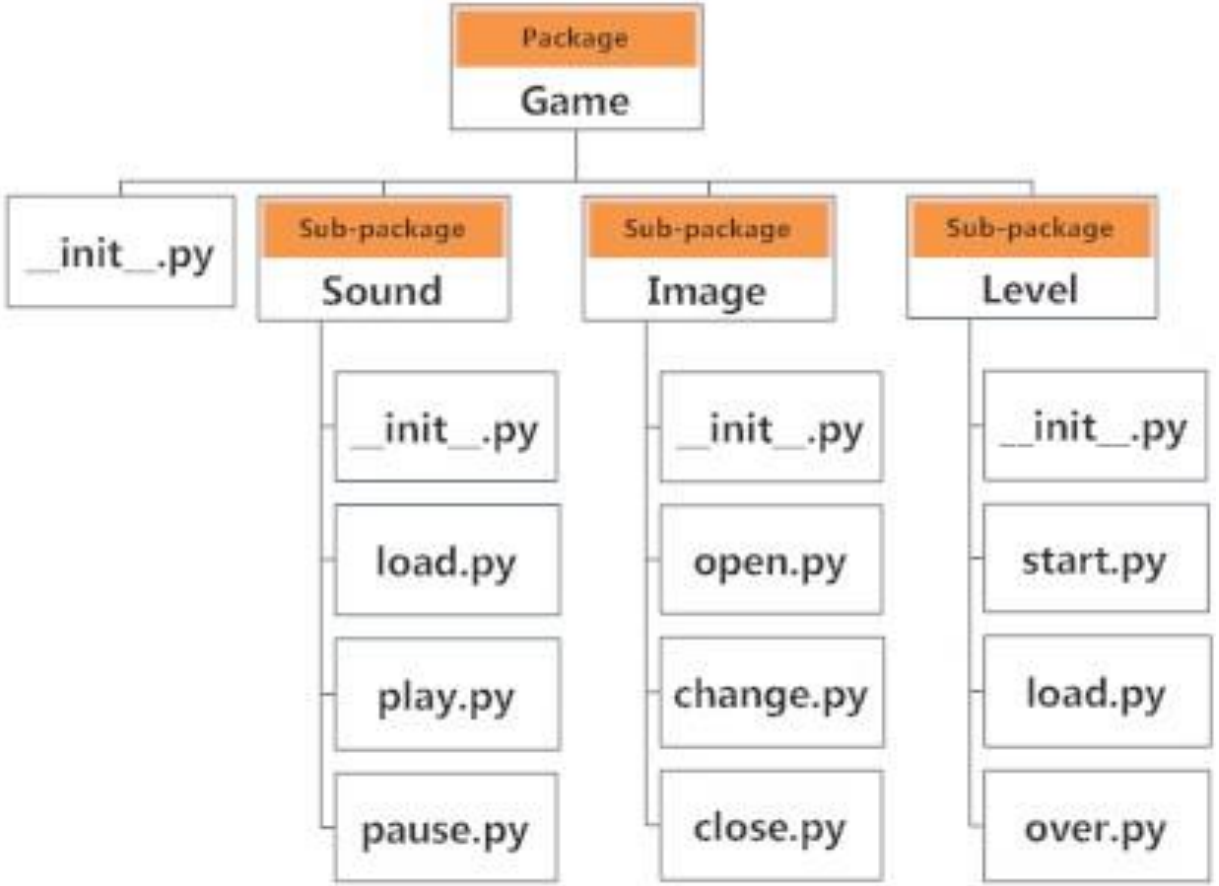
```
# Python Module example
def add(a, b):
        """This program adds two numbers and return the result"""
        result = a + b
        return result
```

- Then, we can import to another module or the interactive interpreter in Python.

```
>>> import example
>>> example.add(4,5.5)
9.5
```

III.8. Package

- Python has packages for directories and modules for files.



III.8. Package

- We can import modules from packages using the dot (.) operator.

```
import Game.Level.start
 Game.Level.start.select_difficulty(2)
```

```
from Game.Level import start
start.select_difficulty(2)
```

```
from Game.Level.start import select_difficulty
select_difficulty(2)
```