**LECTURE-06 (Saturday 17-May-2025)**

* <https://github.com/panaversity/learn-modern-ai-python/blob/main/00_python_colab/08_modules_functions/Agentic_AI_Python_Lesson_08_Module_%26_Functions.ipynb>
* A function or method depends on three things:

1. Declaration (before colon syntax of function)
2. Body (after colon syntax of function)
3. Function calling

* Shortcut key “Ctrl + M + L” is used to show and hide line number.
* Return function: function which returns any value.

def my\_function(a : int, b : int) -> int: (function declaration)

  return a+b (function body)

my\_function(10,20) (function calling)

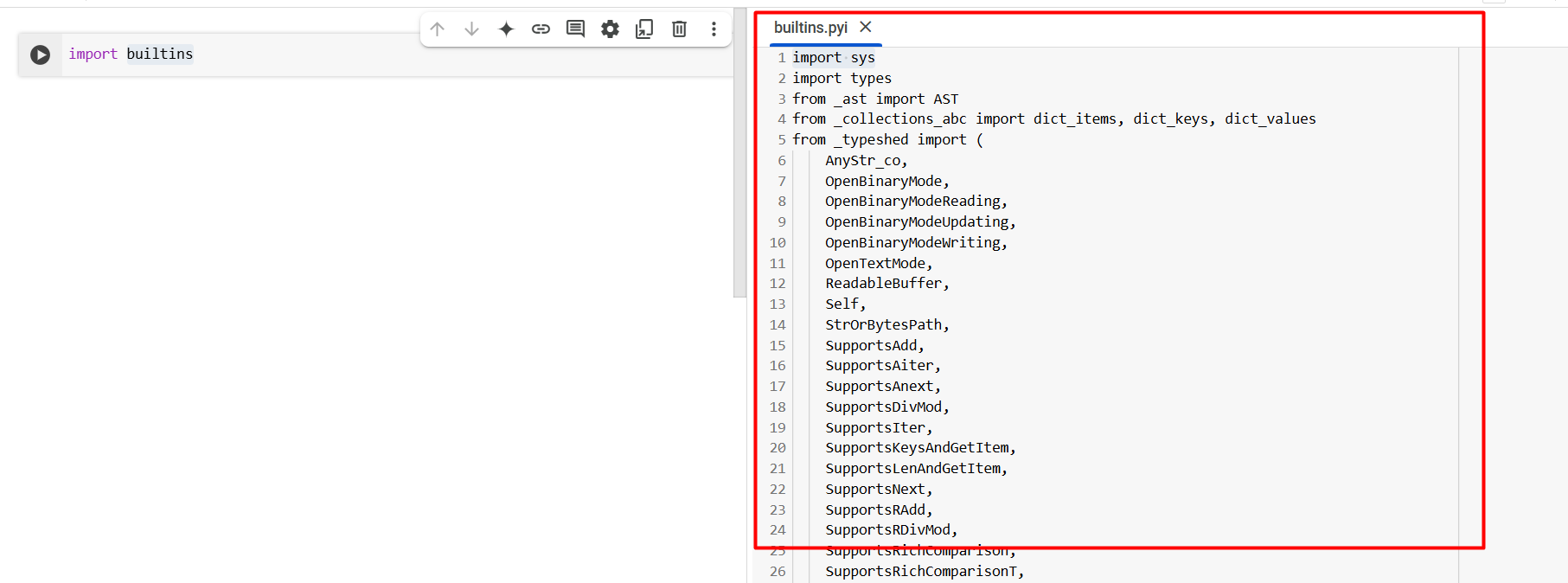
* Built-in functions of python (<https://docs.python.org/3/library/functions.html>)

Or you can check built-in functions of python by:

Write in code cell:

import builtins

Then keep cursor at “builtins” and press “F12” key.



“builtin.pyi” contains all python default functions.

* Important points related to function:

1. what function take (input or parameters)
2. what function returns (output or return value)

* Brainstorming: analyzing input and output of a function
* License function: this function returns python license information

license()

* Multiple Return Types Function: this function supports returning one value of different datatypes.

Example:

def my\_function(a : int, b : int) -> int | float:

  return a + b

abc : int = my\_function(10, 12)

pqr : float = my\_function(10.5, 12.8)

print(abc)

print(pqr)

Output:

22

23.3

* Cursor is like a worker, you give a task (e.g.: SQL query), and it performs every step needed to complete the task like processing, fetching data and finally giving you the result.
* Function name is very important

Example:

def add\_two\_numbers(a : int, b : int) -> int:

  return a + b

* Default function means without parameters function or function which do not require parameters. (e.g.: license)
* Purpose of function or method in programming language:

1. To reduce code repetition
2. For reusability
3. For logic integrity means change in method will affect changes for all places where method is called

* Default or non-parameterized function: function which do not contains parameters

Example:

def piaic\_introduction() -> None:

  print("PIAIC")

  print("Sindh Boys Scouts")

  print("Karachi")

piaic\_introduction()

piaic\_introduction()

piaic\_introduction()

Output:

PIAIC

Sindh Boys Scouts

Karachi

PIAIC

Sindh Boys Scouts

Karachi

PIAIC

Sindh Boys Scouts

Karachi

* Parameterized function: function which contains parameters and these parameters are required to pass as function argument when calling function.

Example:

def add\_two\_numbers(a : int, b : int) -> int:

  return a + b

add\_two\_numbers(10, 12)

Output:

22

* Class Task: Validate provided input is admin credentials

Code:

def is\_valid\_admin\_credentials\_provided(username : str, password : str) -> bool:

  if(username == "admin" and password == "admin"):

    return True

  else:

    return False

allowed\_attempts : int = 3

attempt\_counter = 1

while (attempt\_counter <= allowed\_attempts):

  username : str = input("Enter username :")

  password : str = input("Enter password :")

  if(is\_valid\_admin\_credentials\_provided(username, password)):

    print("Admin access granted")

    break;

  else:

    print("Invalid credentials")

    print(f"Attempt {attempt\_counter} failed while {allowed\_attempts} attempts are allowed.")

    attempt\_counter += 1

Output:

Enter username :a

Enter password :b

Invalid credentials

Attempt 1 failed while 3 attempts are allowed.

Enter username :a

Enter password :b

Invalid credentials

Attempt 2 failed while 3 attempts are allowed.

Enter username :a

Enter password :b

Invalid credentials

Attempt 3 failed while 3 attempts are allowed.

OR

Enter username :a

Enter password :b

Invalid credentials

Attempt 1 failed while 3 attempts are allowed.

Enter username :c

Enter password :d

Invalid credentials

Attempt 2 failed while 3 attempts are allowed.

Enter username :admin

Enter password :admin

Admin access granted

* Function with optional parameters: IF we provided optional parameters then function works with provided values of optional parameters well ELSE function works with default values of optional parameters. One thing, we want to make sure is that values should be passed in sequence to function.

Example:

def add\_numbers(num1 : int, num2: int, num3 : int = 5, num4 : int = 3):

  return num1 + num2 + num3 + num4

add\_numbers(10,4)

Output:

22

OR

def add\_numbers(num1 : int, num2: int, num3 : int = 5, num4 : int = 3):

  return num1 + num2 + num3 + num4

add\_numbers(10,4,8,9)

Output:

31

* Parameter : when creating function while argument: when calling function

Example:

def add\_numbers(num1 : int, num2: int, num3 : int = 5, num4 : int = 3):

  return num1 + num2 + num3 + num4

add\_numbers(10,4)

Here, “num1”, “num2”, “num3” and “num4” are parameters while “10” and “4” are arguments.

* Types of function arguments:

1. Positional argument: arguments are passed in proper position

Example:

def add\_numbers(num1 : int, num2: int, num3 : int = 5, num4 : int = 3):

  return num1 + num2 + num3 + num4

add\_numbers(10,4,8,9)

Here, passing “10,4,8,9” are positional arguments.

1. Keyword arguments: arguments are passed with variable names which are defined in function parameters. In this case, sequence of passing arguments according to sequence of parameters, does not matters.

Example:

def add\_numbers(num1 : int, num2: int, num3 : int = 5, num4 : int = 3):

  return num1 + num2 + num3 + num4

add\_numbers(num2 = 20, num1 = 40, num4 = 80)

Output:

145

* Passing list elements in method arguments

Example:

def add\_numbers(num1 : int, num2: int, num3 : int = 5, num4 : int = 3):

  return num1 + num2 + num3 + num4

data = [10,20]

add\_numbers(data[0], data[1], 30)

Output:

63

* Zip function: It is used to combine list datatype variables.

Example:

k1 = [1,2,3]

k2 = ['a','b','c']

list(zip(k1,k2))

Output:

[(1, 'a'), (2, 'b'), (3, 'c')]

OR

k1 = [1,2,3]

k2 = ['a','b','c']

k3 = [5.67,'b','c']

list(zip(k1,k2,k3))

Output:

[(1, 'a', 5.67), (2, 'b', 'b'), (3, 'c', 'c')]

* Unzip or unpack approach: In this approach, we set value of list elements in different variables.

Example 1: Unzip list elements into variables

a , b, c = [1,2,3]

print("a = ",a)

print("b = ",b)

print("c = ",c)

Output:

a = 1

b = 2

c = 3

Here, error raise if

a , b, c = [1,2]

OR

a , b = [1,2,3]

Example 2: Unzip list elements in function arguments

def my\_function(num1 : int, num2 : int):

  print(f"number1 : {num1} number2 : {num2}")

  return num1 + num2

data = [25,76]

my\_function(\*data)

Output:

number1 : 25 number2 : 76

101

Here, we pass “\*data” in argument.

(\*) is used to unzip list.

Example 3: Unzip dictionary elements in function arguments

def my\_function(num1 : int, num2 : int):

  print(f"number1 : {num1} number2 : {num2}")

  print(num1 + num2)

data = {"num1" : 25, "num2" : 76}

my\_function(data["num1"], data["num2"])

my\_function(\*\*data)

Output:

number1 : 25 number2 : 76

101

number1 : 25 number2 : 76

101

Here, we pass “\*\*data” in argument.

(\*\*) is used to unzip dictionary.

It is important that “key” in dictionary should be same as parameter name of method. In above example:

data = {"num1" : 25, "num3" : 76}

Error raise because dictionary element key is “num3” while parameter variable name of function is “num2”

Also, number of elements in dictionary should be same as number of parameters of function. In above example:

data = {"num1" : 25, "num2" : 76, "num3" : 76}

Error raised because dictionary contains “num3” element but there is no parameter in function with name “num3”.

OR

data = {"num1" : 25}

Error raised because dictionary do not contains “num2” element but there is parameter in function with name “num2”.

* Unlimited list arguments: It means that there is no limit of passing number of list type element arguments in function call

Example:

def my\_sum(num1 : int, \*nums : int) -> int:

  return num1 + sum(nums)

my\_sum(1,2,3,4,5,6,7)

Output:

28

Here, “\*nums” indicates that we can pass unlimited arguments

* Unlimited keyword arguments: It means that there is no limit of passing number of keyword arguments in function call

Example:

def my\_sum(\*\*nums : int) -> None:

  print(nums, type(nums))

my\_sum(a = 1, c= 2.5,  name= "abc")

Output:

{'a': 1, 'c': 2.5, 'name': 'abc'} <class 'dict'>

* Class Task : create “my\_function” where

“num1” and “num2” are required parameters

“num3” is optional parameter

“nums” is unlimited list argument

“nums2” is unlimited keyword argument

Code:

def my\_function(num1 : int, num2 : int, num3 : str = "wow", \*nums1 : int, \*\*nums2) -> None:

    print(num1 + num2)

    print(num3)

    print(nums1)

    print(nums2)

my\_function(10, 20, 5,6,7,8, num4=9, num5=10, num6 = "abc")

Output:

30

5

(6, 7, 8)

{'num4': 9, 'num5': 10, 'num6': 'abc'}

OR

def my\_function(num1 : int, num2 : int, num3 : str = "wow", \*nums1 : int, \*\*nums2) -> None:

    print(num1 + num2)

    print(num3)

    print(nums1)

    print(nums2)

my\_function(10, 20, 5,6,7,8, num4=9, num5=10, num6 = "abc")

nums : list = [8,9,10]

my\_function(10, 20, \*nums, num4=9, num5=10, num6 = "abc")

Output:

30

5

(6, 7, 8)

{'num4': 9, 'num5': 10, 'num6': 'abc'}

30

8

(9, 10)

{'num4': 9, 'num5': 10, 'num6': 'abc'}

OR

def my\_function(num1 : int, num2 : int, num3 : str = "wow", \*nums1 : int, \*\*nums2) -> None:

    print(num1 + num2)

    print(num3)

    print(nums1)

    print(nums2)

my\_function(10, 20, 5,6,7,8, num4=9, num5=10, num6 = "abc")

nums : list = [8,9,10]

my\_function(10, 20, \*nums, num4=9, num5=10, num6 = "abc")

my\_function(10, 20)

Output:

def my\_function(num1 : int, num2 : int, num3 : str = "wow", \*nums1 : int, \*\*nums2) -> None:

    print(num1 + num2)

    print(num3)

    print(nums1)

    print(nums2)

my\_function(10, 20)

Code by Sir:

def my\_function(num1: int, num2 : int, num3 : int = 0, \*skills, \*\*languages\_with\_score):

  print(num1, num2, num3)

  print(skills)

  print(languages\_with\_score)

my\_function(1,2,20, 'web','mobile', 'cloud', python =100, java = 90)

Output:

1 2 20

('web', 'mobile', 'cloud')

{'python': 100, 'java': 90}

OR

def my\_function(num1: int, num2 : int, num3 : int = 0, \*skills, \*\*languages\_with\_score):

  print(num1, num2, num3)

  print(skills)

  print(languages\_with\_score)

my\_function(1,2,20, 'web','mobile', 'cloud', python =100, java = 90)

Output:

1 2 20

('web', 'mobile', 'cloud')

{'python': 100, 'java': 90}

Here, we cannot use dictionary variable to pass unlimited keyword argument

* Lambda function:

Online function

Without name function

Function which is used only once

No name

One line

Not use before

Not use after

def my\_function(num1 : int, num2 : int) -> int:

  return num1 + num2

Above function converted to lambda function as

xyz = lambda num1, num2 : num1 + num2

Example 2:

Prompt: create dictionary and sort with lambda function

Returned Code:

# prompt: create dictionary and sort with lambda function

my\_dict = {'c': 3, 'a': 1, 'b': 2}

# Sort by keys

sorted\_by\_key = dict(sorted(my\_dict.items()))

print("Sorted by key:", sorted\_by\_key)

# Sort by values using a lambda function

sorted\_by\_value = dict(sorted(my\_dict.items(), key=lambda item: item[1]))

print("Sorted by value:", sorted\_by\_value)

Output:

Sorted by key: {'a': 1, 'b': 2, 'c': 3}

Sorted by value: {'a': 1, 'b': 2, 'c': 3}

Here,

“my\_dict.items()” converted dictionary into tuple.

“sorted()” is used to create a new sorted list from an iterable elements of “my\_dict”. It does not modify the original iterable elements of “my\_dict”.

* Generator function:

Run iteratively then stop

Always remember last generated number

Resume iteration on next step

If number is according to our requirement then load number in memory else do not load number in memory, therefore, memory efficient

Example 1:

def my\_range(start : int, end : int, step : int = 1):

    for i in range(start, end + 1, step):

        yield i

numbers = my\_range(1,10)

[i for i in numbers if i%2 == 0]

Ouptut

[2, 4, 6, 8, 10]

Here, other numbers (1, 3, 5, 7, 9) are not loaded in memory.

In Python, `yield` is used in generator functions. Instead of returning a value and exiting the function, a generator function uses `yield` to produce a sequence of values over time. When “yield” is encountered, the function's state is saved, and the yielded value is returned. The next time the generator's “\_\_next\_\_()” method is called, execution resumes from where it left off.

In above example, “yield i” means that for each value “i” in the “range” generated by Python's built-in “range” function, the “my\_range” function will yield that value. This makes `my\_range` a generator function that behaves similarly to Python's built-in “range” function but it generates values on demand rather than creating the entire sequence in memory at once.

Example 2:

def my\_range(start : int, end : int, step : int = 1):

    for i in range(start, end + 1, step):

        yield i

numbers = my\_range(1,10)

print(next(numbers))

print(next(numbers))

print(next(numbers))

print("wow")

print(next(numbers))

print(next(numbers))

print(next(numbers))

print(next(numbers))

print(next(numbers))

print(next(numbers))

Output:

1

2

3

wow

4

5

6

7

8

9

Here, “next” function is used to retrieve the next item from an iterator.

In the provided code, “my\_range(1, 10)” is a generator function which, when called, returns a generator object (an iterator). This generator object, assigned to the variable “numbers”, yields values one by one when the “next()” function is called on it. Each call to “next(numbers)” will get the next value generated by the “my\_range” function’s “yield i” statement, starting from the beginning of the range (1 in this case). The state of the generator is saved between `next()` calls, so it continues from where it left off.

Example 3:

def my\_range(start : int, end : int, step : int = 1):

    for i in range(start, end + 1, step):

        yield i

numbers = my\_range(1,100)

[i for i in numbers if i%3 == 0 and i%5 == 0 ]

Output:

[15, 30, 45, 60, 75, 90]

* In other programming languages, “Array”:

Fixed length

Same datatype elements

* Recursive function:

Function call itself inside its body

We have to break recursion at some point

Recursion is like loop

Example:

def factorial(n):

  print(n, n-1)

  if n==1:

    return 1

  else:

    return n \* factorial(n-1)

print(factorial(5)) # factorial of 5 = 5 x 4 x 3 x 2 x 1 = 120

Output:

5 4

4 3

3 2

2 1

1 0

120

* Decorator function: Any function above which another function is present and starts with “@” sign.

Example:

def hello\_fun() -> str:

  return "Hello world"

def my\_decorator(fun):

  print("start line")

  print(fun)

  print("end line")

my\_decorator(hello\_fun())

hello\_fun()

Output:

start line

Hello world

end line

OR

def my\_decorator(fun):

  def wrapper():

    print("start line")

    fun()

    print("end line")

  return wrapper

@my\_decorator

def hello\_fun() -> str:

   print("Hello world")

hello\_fun()

Output:

start line

Hello world

end line

* Prompt: generate modern decorator function

Returned code:

# prompt: generate modern decorator function

def my\_decorator(func):

  def wrapper():

    print("Something is happening before the function is called.")

    func()

    print("Something is happening after the function is called.")

  return wrapper

@my\_decorator

def say\_hello():

  print("Hello!")

say\_hello()

Output:

Something is happening before the function is called.

Hello!

Something is happening after the function is called.