



COMP 1023 Introduction to Python Programming Functions (Part I)

Dr. James Cheung, Dr. Wilfred Ng, Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology, Hong Kong SAR, China



Introduction

- Suppose that we need to find the sum of integers from 1 to 10, from 20 to 37, and from 35 to 49.

```
# Filename: sum_of_integers.py
def main():
    total = 0
    for i in range(1, 11):
        total += i
    print("Sum of integers from 1 to 10 is", total)

    total = 0
    for i in range(20, 38):
        total += i
    print("Sum of integers from 20 to 37 is", total)

    total = 0
    for i in range(35, 50):
        total += i
    print("Sum of integers from 35 to 49 is", total)

if __name__ == "__main__":
    main()
```

Observations

The code for computing these sums is **very similar**, except that the **starting and ending integers** are different.



Introduction

- It is better to **write commonly used code once** and then **reuse it**.
- In Python and other programming languages, we can do this by **defining a function**, which **enables us to create reusable code**.
- For example, the preceding code can be simplified by using functions, as follows:

```
# Filename: sum_of_integers_func.py
```

```
def sum_range(i1, i2):  
    result = 0  
    for i in range(i1, i2 + 1):  
        result += i  
    return result  
  
def main():  
    print("Sum of integers from 1 to 10 is", sum_range(1, 10))  
    print("Sum of integers from 20 to 37 is", sum_range(20, 37))  
    print("Sum of integers from 35 to 49 is", sum_range(35, 49))  
  
if __name__ == "__main__":  
    main()
```



Functions

- A **function** is a collection of statements grouped together that performs an operation. For example, `input("Enter a value ")` is a function.

Syntax

```
def <function_name>(<parameter_list>):  
    <statement_1>  
    <statement_2>  
    ...  
    <statement_N>
```

where `<function_name>` is the name of the function, `<parameter_list>` is a list of parameters (i.e., a number of variables), and `<statement_1>`, ..., `<statement_N>` denote the program statements that need to be executed when the function is called/invoked.

- Now, you should notice that `'main'` is actually a function, which serves as the entry point of a program.

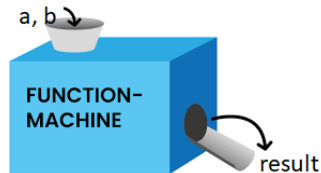
Analogy of Python Functions: The Machine Model

- Think of a **Python function** as a **machine that performs specific tasks**.
- The function accepts **inputs**, **processes** them, and produces **outputs**:
 - **Inputs**: These are the **arguments or parameters** you provide to the function, similar to raw materials fed into a machine.
 - **Processing**: The function executes a **series of operations** on the inputs, akin to the machine performing its designated tasks.
 - **Outputs**: The **result produced** by the function after processing, comparable to the finished product that comes out of the machine.
- Example:

```
# Filename: add_numbers.py
```

```
def add(a: int, b: int) -> int:  
    return a + b      # The machine adds two numbers and  
                     # returns the result
```

```
result: int = add(5, 3) # Inputs: 5 and 3; Output: 8
```



Terminologies

```
# Filename: maximum_two_numbers.py

# Define a function
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2
    return result

def main():
    x, y = 10, 20
    z = max(x, y) # Call the function
    print("The larger number is", z)

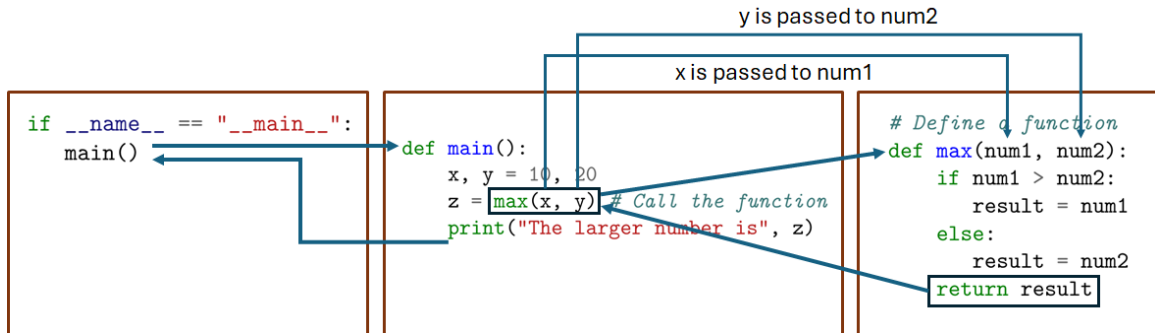
if __name__ == "__main__":
    main()
```

Output:

The larger number is 20

- Function name: **max**
- Formal parameters: **num1 and num2**
- Parameter list: **num1, num2**
- Function header:
def max(num1, num2)
- Function body: **The function body implements the logic of the function.**
- Where does the function return a value? **The return statement returns the value, i.e.,**
return result
- Function caller of max: **main()**
- Actual parameters (or arguments): **x and y**

Program Flow

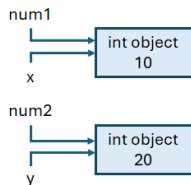


- When a function is called/invoked, control is transferred to the function.
- When the function is finished, control is returned to the point where the function was called.

Parameter Passing

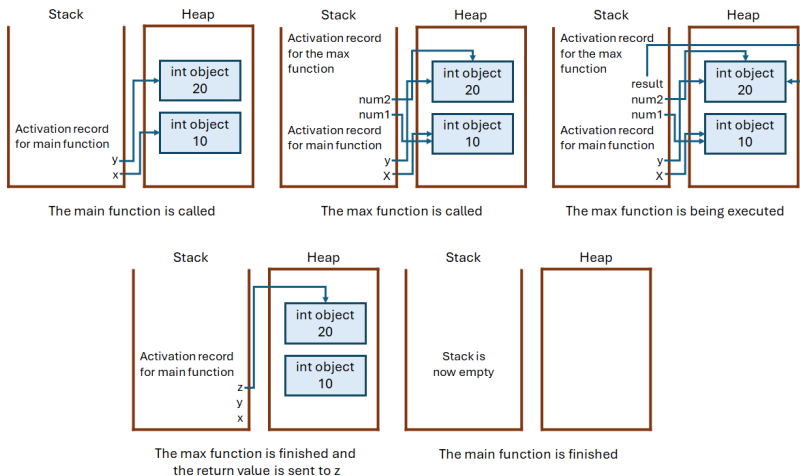
```
# Define a function
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2
    return result
```

```
def main():
    x, y = 10, 20
    z = max(x, y) # Call the function
    print("The larger number is", z)
```



- Variable `x` references an integer object with the value 10. Passing `x` to `num1` is actually passing the reference of the object to `num1`, so `x` and `num1` point to the same object if value does not change.
- Similarly, variable `y` references an integer object with the value 20. Passing `y` to `num2` is actually passing the reference of the object to `num2`, so `y` and `num2` point to the same object if value does not change.

Activation Records



- Activation records (or stack frames) are used to store information about active functions.
- They contain details such as function parameters, local variables, and return addresses.
- Each time a function is called, a new activation record is created on the call stack.

Type Hinting in Functions

- Recall, **type hinting** allows you to **specify the expected data types of parameters** (now function parameters). You can also specify what the **expected types of return values are**.
- It enhances code readability and helps with static type checking.

Syntax

- Use a colon (:) to annotate parameter types.
- Use an arrow (->) to annotate the return type.

```
def max(num1: int, num2: int) -> int:  
    if num1 > num2:  
        result: int = num1  
    else:  
        result: int = num2  
    return result
```

Benefits of Type Hinting

- Improves code clarity and documentation.
- Facilitates easier debugging and maintenance.

None Function

- Technically, every function in Python returns a value, whether you use a return statement or not.
- If a function does not return a value, by default, it returns a special value, None.
- A function that does not return a value is called a None function or a void function.
- A return statement is not needed for a None/void function, but it can be used to terminate the function and return control to the caller. The syntax is return or return None.

```
def sum(number1: int, number2: int) -> None:
    total: int = number1 + number2
    # No return statement

def main() -> None:
    print(sum(1, 2)) # Print None

if __name__ == "__main__":
    main()
```

```
def sum(number1: int, number2: int) -> None:
    total: int = number1 + number2
    return # Explicitly returning None
           # Alternative: return None

def main() -> None:
    print(sum(1, 2)) # Print None

if __name__ == "__main__":
    main()
```

Ordering of the Functions

- In the last example, 'main' is defined after 'max'. Can 'main' be defined before 'max'?
Yes! In Python, **functions can be defined in any order** in a .py file, as a function is loaded into memory when it is called.

```
# max function before main
def max(num1: int, num2: int) -> int:
    if num1 > num2:
        result: int = num1
    else:
        result: int = num2
    return result

def main() -> None:
    x: int = 10
    y: int = 20
    z: int = max(x, y) # Call the function
    print("The larger number is", z)

if __name__ == "__main__":
    main()
```

```
# main function before max
def main() -> None:
    x: int = 10
    y: int = 20
    z: int = max(x, y) # Call the function
    print("The larger number is", z)

def max(num1: int, num2: int) -> int:
    if num1 > num2:
        result: int = num1
    else:
        result: int = num2
    return result

if __name__ == "__main__":
    main()
```

Positional and Keyword Arguments

- When calling a function, we need to pass arguments to the formal parameters.
- There are two kinds of arguments: **positional arguments** and **keyword arguments**.
- Using **positional arguments** requires that the arguments be passed in the same order as their respective parameters in the function header.
- We can also call a function using **keyword arguments**, passing each argument in the form **name = value**.

```
# Filename: print_n_times.py
def print_n_times(text: str, n: int) -> None:
    for i in range(n):
        print(text)

def main() -> None:
    # Pass "COMP 1023" to text, and 3 to n
    print_n_times("COMP 1023", 3) # Print COMP 1023 3 times
    # Can we do print_n_times(3, "COMP 1023")? ...
    # Pass 3 to n, and "COMP 1023" to text
    print_n_times(n=3, text="COMP 1023") # Print COMP 1023 3 times

if __name__ == "__main__": main()
```

Output:

```
COMP 1023
COMP 1023
COMP 1023
COMP 1023
COMP 1023
COMP 1023
```

Positional and Keyword Arguments

- It is possible to mix positional arguments with keyword arguments, but positional arguments cannot appear after any keyword arguments.

```
# Filename: sum.py
```

```
def sum(value1: int, value2: int, value3: int) -> int:  
    return value1 + value2 + value3
```

```
def main() -> None:  
    print(sum(10, value2=20, value3=30)) # Print 60  
    print(sum(10, value2=20, 30)) # Error: positional argument follows keyword argument
```

```
if __name__ == "__main__":  
    main()
```



What do you observe?

```
# Filename: increment.py
def increment(n: int) -> None:
    n += 1
    print("n inside the function is", n)

def main() -> None:
    x: int = 1
    print("Before the call, x is", x)
    increment(x)
    print("After the call, x is", x)

if __name__ == "__main__": main()
```

Output:

Before the call, x is 1
n inside the function is 2
After the call, x is 1



Observations

- Integers in Python are immutable. When you pass an integer to a function, you are passing a reference to the value, not the variable itself. Any modifications to the parameter inside the function do not affect the original variable outside the function.
- To modify the original variable, you would need to return the new value from the function and reassign it to the original variable, or use a mutable type (like a list or dictionary) to hold the value.

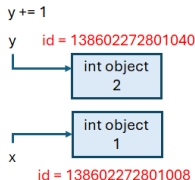
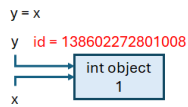
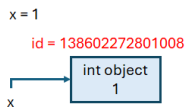
Immutable Objects

- Numbers are **immutable objects**. The content of **immutable objects** cannot be changed.
- Whenever we **assign a new number to a variable**, Python **creates a new object** for the new number and **assigns the reference of the new object to the variable**.

```
x: int = 1
y: int = x
print(id(x))  # The reference of x
print(id(y))  # The reference of y
y += 1
print(id(y))  # The reference of y
```

Output:

```
138602272801008
138602272801008
138602272801040
```



The `id(object)` function returns the unique id of an object.

Default Arguments

- Python allows you to define functions with **default argument values**.
- The **default values are used for the parameters** when a function is called **without specific arguments**.

```
def print_area(width: float = 1, height: float = 2) -> None: # Filename: print_area.py
    area: float = width * height
    # \t is an escape sequence that represents a tab character.
    print("Width:", width, "\theight:", height, "\tarea:", area)

def main() -> None:
    print_area() # Default arguments with width = 1, height = 2
    print_area(4, 2.5) # Positional arguments with width = 4, height = 2.5
    print_area(height=5, width=3) # Keyword arguments
    print_area(10) # Positional argument with width = 10, height = 2
    print_area(width=1.2) # Keyword argument with width = 1.2, default height = 2
    print_area(height=6.2) # Keyword argument with height = 6.2, default width = 1

if __name__ == "__main__":
    main()
```

Default Arguments

Output:

Width: 1	height: 2	area: 2
Width: 4	height: 2.5	area: 10.0
Width: 3	height: 5	area: 15
Width: 10	height: 2	area: 20
Width: 1.2	height: 2	area: 2.4
Width: 1	height: 6.2	area: 6.2



Default Arguments

Rule

Default arguments must be placed after required arguments.

- The following function will raise a syntax error because default arguments cannot come before required ones:

```
def print_area(height: float = 2, width: float) -> None: # Error!  
    area: float = width * height
```

- The following function is correct, but the last call will cause an error because the width is required but not provided:

```
def print_area(width: float, height: float = 2) -> None:  
    area: float = width * height  
    print("Width:", width, "\theight:", height, "\tarea:", area)
```

```
def main() -> None:  
    print_area(4)           # width = 4, height = 2 (default)  
    print_area(4, 5)        # width = 4, height = 5  
    print_area(height=6)     # width is missing, height = 6 (error)
```

Returning Multiple Values

- Python allows a **function** to **return multiple values**.
- The function, `sort`, takes two numbers and returns them in ascending order.

Filename: sort_numbers.py

```
def sort(number1: int, number2: int) -> tuple[int, int]:  
    if number1 < number2:  
        return number1, number2  
    else:  
        return number2, number1
```

```
def main() -> None:  
    n1: int; n2: int  
    n1, n2 = sort(3, 2)  
    print("n1 is", n1)  
    print("n2 is", n2)
```

```
if __name__ == "__main__": main()
```

Output:

n1 is 2
n2 is 3



- The **values returned** by the `sort` function are **packed into a tuple**.
- This tuple can be **unpacked directly into multiple variables**, making it convenient to work with multiple return values.

pass Statement

- The `pass` statement is a **null operation**. When it is executed, **nothing happens**.
- It is used as a **placeholder** where syntactically some code is required, but no action is needed.
- Uses of the `pass` statement:
 - Empty function bodies
 - Empty classes (to be discussed later)
 - Selection/branching statements
 - Looping/iterative statements



Uses of pass Statement

- **Empty function bodies:** When defining a **function**, Python requires some code within the function block. If you **don't have any idea yet**, you can use **pass** to avoid syntax errors.

```
def my_function() -> None:  
    pass
```

- **Empty classes:** Similar to functions, **classes** in Python also need a body. You can use **pass** to **create an empty class**. (We will talk more about this later.)

```
class MyClass:  
    pass
```



- **Selection/branching statements:** In **if**, **elif**, or **else** blocks, if you want to **do nothing under certain conditions**, **pass** can be used.

```
x: int = 5  
if x > 10:  
    pass # No action if x > 10  
else:  
    print("x is not greater than 10")
```

- **Looping/iterative statements:** Similarly, in loops, you can use **pass** if you intend to **do nothing within the loop under specific conditions**.

```
for i in range(5):  
    if i == 3:  
        pass # Skip action when i is 3  
    else:  
        print(i)
```

Key Terms

- actual parameter
- argument
- caller
- default argument
- formal parameter (i.e., parameter)
- function
- function header
- immutable objects
- keyword arguments
- local variable
- None
- None function
- parameter
- positional arguments
- return value
- void function

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

1. A function header begins with the _____ keyword followed by the _____ and its _____, and ends with a colon.
2. A function is called a _____ if it does not return a value.
3. A _____ statement can also be used in a void function for terminating the function and returning to the function's caller.
4. The _____ that are passed to a function should have the same _____, _____, and _____ as the parameters in the function header if no default values or keywords specified.

Answer: 1. def; function's name; parameters, 2. void function, 3. return, 4. arguments; number; type; order.

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

5. A function's arguments can be passed as _____ or _____.
6. Python allows you to define functions with _____. The _____ are passed to the parameters when a function is invoked without the arguments.
7. The Python return statement can return _____.

Answer: 5. positional arguments; keyword arguments, 6. default argument values; default values, 7. multiple values

Further Reading

- Read Chapter 6 of “Introduction to Python Programming and Data Structures” textbook.



That's all!

Any questions?

