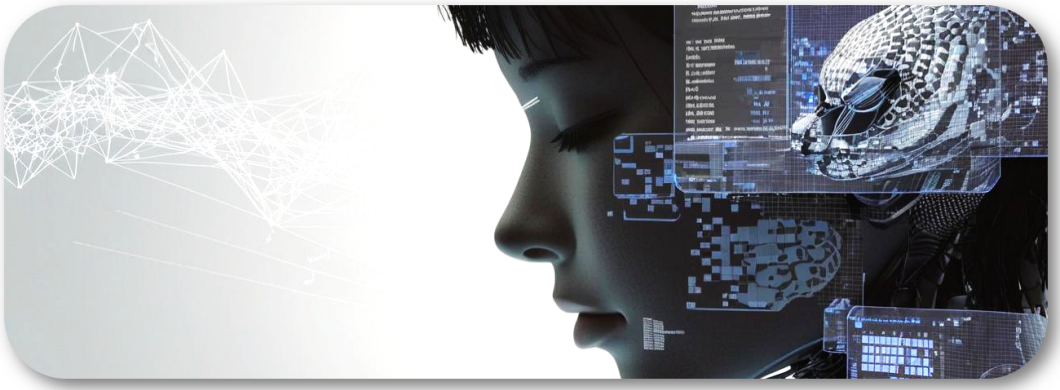

COMP 1023 Introduction to Python Programming

Tutorial: Functions

COMP 1023 Teaching Team

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

Quick Refresher - Functions

A function is a collection of statements grouped together that performs an operation.

```
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.

Example

```
def add_and_multiply(x, y):
    print(x + y)
    return x + y, x * y
```

Quick Refresher - Type Hinting

Type hinting allows you to specify the expected data types of function parameters You can also specify the expected type of the returned value(s). We utilize type hinting to:

- Enhance code readability, clarity and improve documentation
- Help with static type-checking
- Facilitate easier debugging and maintenance

Example

```
def add_and_multiply(x: int, y: int) -> tuple[int, int]:
    print(x + y)
    return x + y, x * y
```

Quick Refresher - Ordering

A function can be defined in any order in a .py file, as long as a function is loaded into memory when it is called.

```
def main() -> None:
    print(add(1, 2))

def add(x: int, y: int) -> int:
    return x + y

if __name__ == "__main__":
    main() # main is called here
           # No errors are raised since add is already defined
```

iPRS Question 1

Recall what you've learnt in lectures. What should be the expected type of returned value in the following function?

```
def func(x: int, y: list[int]) -> ???:
    v: int = x + y[0] if len(y) > 0 else x
    y.append(v)
```

A. `int`

B. `list[int]`

C. `None`

iPRS Question 1 - Answer

What should be the expected type of returned value in the following function?

A. `int`

B. `list[int]`

C. `None`

Explanation

Functions that do not have an explicit `return` statement returns the special value `None`.
Think about:

```
def func(x: int, y: list[int]) -> ???:
    v: int = x + y[0] if len(y) > 0 else x
    y.append(v)
    return
```

What should be the type now?

Parameter Passing in Functions

- Any variables passed to functions as parameters initially reference the original object
- Note that assigning a different object to the variable will not change the original

```
def example(x: int) -> None:
    x = 0

if __name__ == "__main__":
    a = 1023
    example(a)
    print(a)  # Output is still 1023
```

Practice Question: Parameter Passing in Functions

What is the output of the following program?

```python
def func(x: list[int]) -> list[int]:
    x = [1, 2, 3]
    return x

if __name__ == "__main__":
    nums = [1, 2, 3]
    ret = func(nums)
    print(nums == ret)
    print(nums is ret)
```

Practice Question: Parameter Passing in Functions - Answer

```
True
False
```

Explanation

The `func` function assigns another list `[1, 2, 3]` to `x` before returning it.

That means `nums` and `ret` refer to 2 different lists. Therefore, `==` returns `True` since they have the same values within, but `is` returns `False` as they are not the same `list` object.

Positional and Keyword Arguments

- Positional arguments: Parameters passed in the exact order of function definition
- Keyword arguments: Parameters specified with exact variable name in function definitions
  - Their order does not need to follow declaration order
- When mixing both types of arguments:
  - Positional arguments cannot appear after keyword arguments

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

def main():
    print(calc(10, value3=30, value2=20)) # OK, prints 230
    print(calc(10, value2=20, 30))        # Error

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

The pass Statement, Default Arguments and Multiple Return Values

- The pass statement: A Python statement that means "do nothing"
- Default arguments: Parameter values specified in function definition
- Multiple return values: Returning multiple values via tuple packing

```python
def do_nothing() -> None:
    pass

def calc(a: int, b: int = 20) -> tuple[int, int]:
    return a + b, a - b

if __name__ == "__main__":
    x, y = calc(50)
    do_nothing()               # Me on holidays
    print(f"x: {x}, y: {y}")  # x: 70, y: 30
```

iPRS Question 2

What is the output of the following program?

```
def calc(a: int = 1, b: int = 2, c: int = 3, d: int = 4) -> int:
    return a * 1000 + b * 100 + c * 10 + d

if __name__ == "__main__":
    x = calc(2, b = 5, d = 9)
    print(x)
```

A. No output due to runtime error

B. 1234

C. 1539

D. 2539

iPRS Question 2 - Answer

What is the output of the following program?

```
def calc(a: int = 1, b: int = 2, c: int = 3, d: int = 4) -> int:
    return a * 1000 + b * 100 + c * 10 + d

if __name__ == "__main__":
    x = calc(2, b=5, d=9)
    print(x)
```

A. No output due to runtime error

B. 1234

C. 1539

D. 2539

Explanation

- In function `calc`, a $= 2$, b $= 5$, c $= 3$, d $= 9$
- answer $= 2 \times 1000 + 5 \times 100 + 3 \times 10 + 9 = 2539$

Variable Scopes

- Local variables defined inside functions are contained within the function. They cannot be accessed outside the function in which they're defined.
  - This is known as "local scope".
- Global variables defined outside any functions are contained within the entire program. They can be accessed anywhere, inside or outside functions.
  - This is known as "global scope".
  - In functions, you can not change a global variable's reference unless you bind a local variable to the global scope.
- Local variables can have the same name as global variables.
  - This is known as "variable shadowing".
- To bind a local variable to the global scope, use the `global` keyword.

Variable Scopes - Example

```
x = 1023
def local():
    x = 5
    return x # Returns the local variable

def using_global():
    return x # Returns the global variable

def changing_global():
    global x # Use the "global" keyword to use x for assignment
    x = 5    # Changes the value of the global x to 5

if __name__ == "__main__":
    print(local())            # 5
    print(using_global())     # 1023
    print(changing_global()) # None
    print(x)                  # 5
```

iPRS Question 3

What is the output of the following program?

```
v = 1000
def reset() -> None:
    v = 1000
def increment() -> None:
    global v
    v += 1
def main() -> None:
    for i in range(20, 65, 2):
        reset()
        increment()
    print(v)
if __name__ == "__main__":
    main()
```

A. Runtime error as `main` cannot access `v`.

B. 1000

C. 1001

D. 1023

iPRS Question 3 - Answer

What is the output of the program?

A. Runtime error as `main` cannot access `v`.

B. 1000

C. 1001

D. 1023

Explanation

The `reset` function does not change what the global variable `v` references to, so the global variable `v` is incremented 23 times.

Passing Collections to Functions

- When mutable objects are passed into a function as a parameter, they can be mutated within.

```python
def mutate_list(list_a: list[int]) -> None:
    if len(list_a) > 0:
        list_a[0] = 100

def main() -> None:
    my_list = [1, 2, 3]
    ret = mutate_list(my_list)
    print(ret)      # None
    print(my_list)  # [100, 2, 3]

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

Practice Question: Passing Collections to Functions

What is the output of the following program?

```
def func(x: list[int], y: int) -> list[int]:
    x.append(y)
    x = [i for i in x]
    return x

def main() -> None:
    nums = [5, 4, 3, 2]
    ret = func(nums, 1)
    ret.append(0)
    print(f"nums: {nums}")
    print(f"ret : {ret}")

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

Practice Question: Passing Collections to Functions - Answer

```
nums: [5, 4, 3, 2, 1]
ret : [5, 4, 3, 2, 1, 0]
```

Explanation

The `func` function first appends `y` into `x`. This does not change what `x` refers to. Afterwards, the function creates a shallow copy of `x`, and assigns it to `x` before returning it.

That means `nums` and `ret` now refer to different lists. Therefore, when the program appends 0 to `ret`, `nums` is not changed.

Mutable Objects as Default Parameter Values

- In Python, default values are evaluated once only when the function is first defined.
- That means, if a default value is mutable and you mutate it, you will have that mutated object as the default value for any future calls.
- To avoid this behavior, we should use `None` as the default value.

```python
def default_list(lst: list[int] = []) -> list[int]:
    lst.append(0)
    return lst

def default_none(lst: list[int] | None = None)\
    -> list[int]:
    if lst is None:
        lst = []
    lst.append(0)
    return lst
```

```python
def main():
    print(default_list()) # [0]
    print(default_list()) # [0, 0]
    print(default_none()) # [0]
    print(default_none()) # [0]

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

iPRS Question 4

What is the output of the following program?

```
def func1(lst: list[int] = [], val: int = 1023)\
    -> list[int]:
    lst.append(val)
    return lst

def func2(lst: list[int] | None = None, val: int = 1023)\
    -> list[int]:
    if lst is None:
        lst = []
    lst.append(val)
    return lst
```

```
def main() -> None:
    for _ in range(3):
        func1(val=100)
        func2()
    ret1 = func1()
    ret2 = func2()
    print(ret2 < ret1)

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

A. True

B. False

C. No output due to runtime error

iPRS Question 4 - Answer

What is the output of the program?

A. True

B. False

C. No output due to runtime error

Explanation

In the for-loop, the value `100` is appended to the default empty list 3 times in `func1`. Then, `1023` is appended to it. For `func2`, a new list is created every time, so the results of `func2` in the loop is ignored.

	Value
`ret1`	[100, 100, 100, 1023]
`ret2`	[1023]

Even though the length of `ret1` is longer than `ret2`, '<' compares lists element-by-element. Since `ret2[0]` is larger than `ret1[0]`, `ret2 < ret1` is evaluated to `False`.

Modularizing Code

- We can place function definitions in other .py files, then import them for later use.
- These files are known as "modules".
- They should be placed in the same directory as your other programs.

Modules, Packages and Libraries (Advertisement)

We will explore more about modules, packages and libraries in the next tutorial!

That's all!

Any questions?