Computational Thinking

Lecture 07: Function – Specification – Testing

University of Engineering and Technology
VIETNAM NATIONAL UNIVERSITY HANOI

Outline

- Function
- Module
- Specification
- Testing

Outline

- Function
- Module
- Specification
- Testing

Python interactive mode

```
>>> x = 5
>>> y = 4
>>> bigger = max(x, y)
>>> bigger
5
>>> pi = 3.14159
>>> a = round(pi, 2)
>>> a
3.14
```

Executes assignment statements

Evaluates the expression and displays value

Executes assignment statements

Evaluates the expression and displays value

Python interactive mode displays the value to be helpful

Python built-in functions

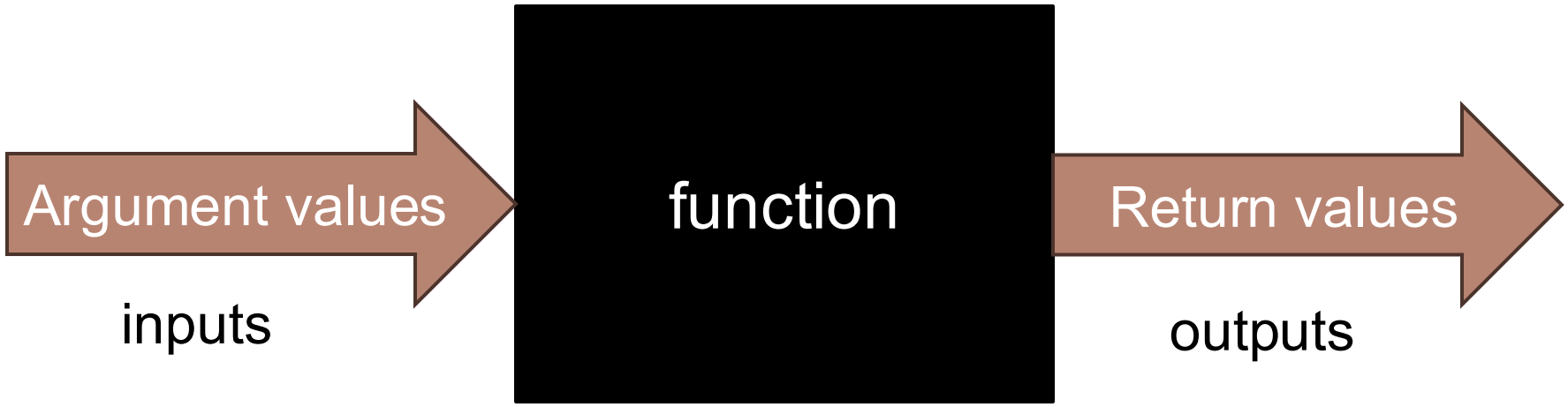
```
>>> x = 5
>>> y = 4
>>> bigger = max(x, y)
>>> bigger
5
>>> pi = 3.14159
>>> a = round(pi, 2)
>>> a
3.14
```

Find the larger number

Limit a number to 2 decimal places

max() and round() are two built-in functions in Python

Functions are value transformers



Python built-in functions

- **Predefined** by the Python language
- Ready to use for common tasks
  - Type conversion: `int(), float(), bool()`
  - Get type of value: `type()`
  - In/out: `input(), print()`
  - Basic calculations: `max(), min(), abs()`
- Full list:
  - `help(__builtins__)`
  - Python docs

Function documentation

- How do we know what a function does?

→ Read its documentation!

Function documentation

round(*number*, *ndigits=None*)

Return *number* rounded to *ndigits* precision after the decimal point. If *ndigits* is omitted or is `None`, it returns the nearest integer to its input.

For the built-in types supporting `round()`, values are rounded to the closest multiple of 10 to the power minus *ndigits*; if two multiples are equally close, rounding is done toward the even choice (so, for example, both `round(0.5)` and `round(-0.5)` are `0`, and `round(1.5)` is `2`). Any integer value is valid for *ndigits* (positive, zero, or negative). The return value is an integer if *ndigits* is omitted or `None`. Otherwise, the return value has the same type as *number*.

For a general Python object `number`, `round` delegates to `number.__round__`.

Note: The behavior of `round()` for floats can be surprising: for example, `round(2.675, 2)` gives `2.67` instead of the expected `2.68`. This is not a bug: it's a result of the fact that most decimal fractions can't be represented exactly as a float. See Floating-Point Arithmetic: Issues and Limitations for more information.

https://docs.python.org/3/library/functions.html#round

Function name

Function parameters

round(*number*, *ndigits=None*)

Return *number* rounded to *ndigits* precision after the decimal point. If *ndigits* is omitted or is `None`, it returns the nearest integer to its input.

For the built-in types supporting `round()`, values are rounded to the closest multiple of 10 to the power minus *ndigits*; if two multiples are equally close, rounding is done toward the even choice (so, for example, both `round(0.5)` and `round(1.5)` is `2`). Any integer value is valid for *ndigits* (positive, zero, or negative). The return value is an integer if *ndigits* is omitted or `None`. Otherwise, the return value has the same type as *number*.

What the function does

For a general Python object `number`, `round` delegates to `number.__round__`.

> **Note:** The behavior of `round()` for floats can be surprising: for example, `round(2.675, 2)` gives `2.67` instead of the expected `2.68`. This is not a bug: it's a result of the fact that most decimal fractions can't be represented exactly as a float. See Floating-Point Arithmetic: Issues and Limitations for more information.

https://docs.python.org/3/library/functions.html#round

Why functions?

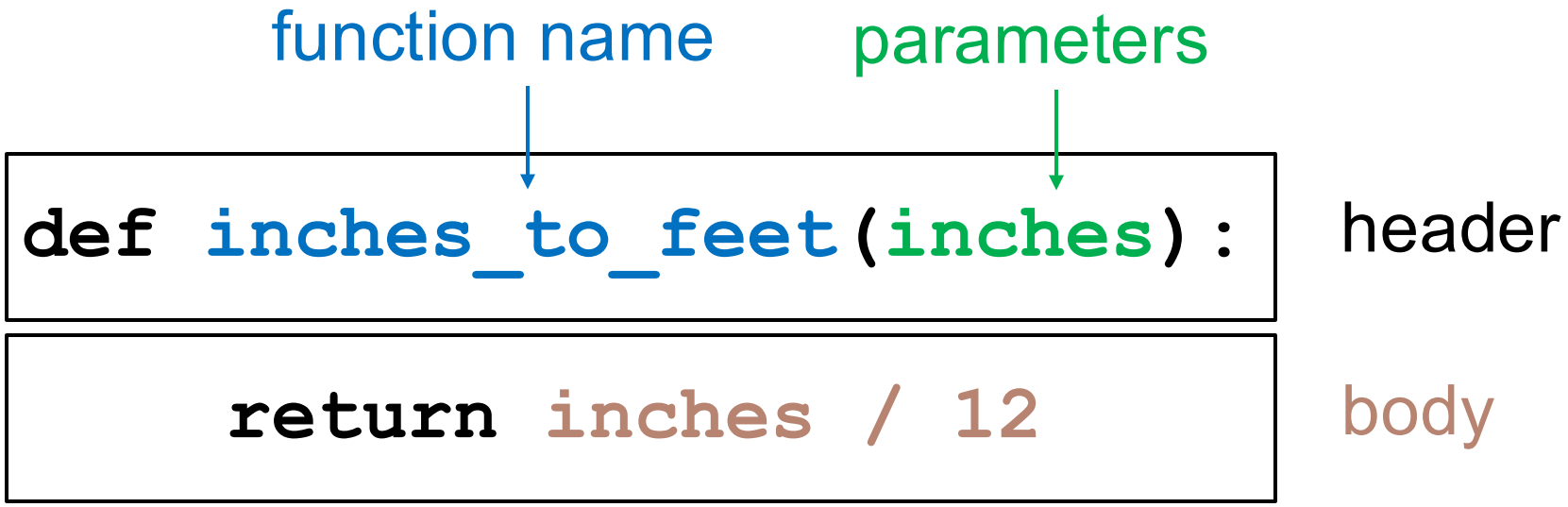
- Organize your code
  - Group related statements under a meaningful name
  - Easy to read and debug
- Eliminate repetition
  - If you make a change, you only have to make it in one place
- Enable code reuse
  - Reuse functions in other programs

Defining your own functions

- Let's define your own function to:

  convert inches to feet

Defining your own functions



- Header: starts with def and ends with colon (:)
- Body: code indented underneath header
- Parameter: a variable named in the header that can be used in the body. It stands for the input value that will be given to the function any time the function is called.

Syntax of function definition

```
def <funct_name>(<param1>, …):
    <statements/expressions>
```

def is a keyword: a “word” that means something special to Python, hence cannot be used as a variable name

Function call vs. function definition

Function call

```
if __name__== "__main__":
    inches_to_feet(65)
```

- Tell Python to execute the function
- Has the arguments in parentheses

Function definition

```
def inches_to_feet(inches):
    return inches/12
```

- Tells Python what to do when execution of a function is requested
- Has the parameters in parentheses

The return statement

- Determines the output value of the function
- Ends the execution of the function
- Syntax: `return <expression>`
- `return` is another keyword

Common questions about return

- Q: What's the difference between `return` and `print()`?
- A: `return` is a statement that moves data around in memory. `print()` is a function that makes symbols appear on the screen.

- Q: What if a function is missing a `return`?
- A: Execution ends with the last line of the function, and a special value None is return

- Q: What if there are function body statements after a `return`?
- A: They are ignored.

Call stack

- When a function is called, Python uses a call stack to keep track of where it is in the program
- Stack structure:
  - Each time a function is called, a new frame is added (pushed) to the stack
  - When the function finishes, the frame is removed (popped) from the stack

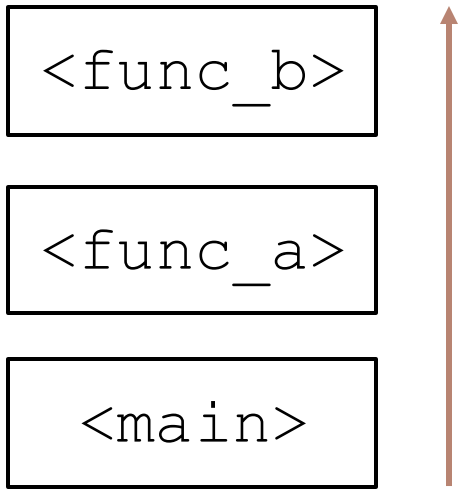
Call stack

```
def func_b():
    print("func b")

def func_a():
    print("func a")
    func_b()

if __name__ == "__main__":
    func_a();
```

Call stack



Local vs. global variables

- Global variables:
  - Defined outside any function
  - Accessible throughout the program
- Local variables:
  - Defined inside a function
  - Only accessible within that function

Access to local vs. global variables

- A function can access:
  - Its own local variables
  - Global variables (read-only by default)
- Assignment to a variable inside a function with the same name as a global one:
  - Creates a new local variable
  - The global remains unchanged
  - To modify a global variable in a function, use `global` keyword
- Global variables are useful as constants

Local vs. global variables

```
counter = 0

def show_counter():
    print("Current counter:", counter)

def bad_increment():
    counter = counter + 1
    print("Bad_increment:", counter)

def good_increment():
    global counter
    counter = counter + 1
    print("Good_increment:", counter)
```

Local vs. global variables

```
counter = 0

def show_counter():
    print("Current counter:", counter)

def bad_increment():
    counter = counter + 1
    print("Bad_increment:", counter)

def good_increment():
    global counter
    counter = counter + 1
    print("Good_increment:", counter)
```

counter is a global variable so it is accessiable inside any function

Local vs. global variables

```
counter = 0

def show_counter():
    print("Current

def bad_increment():
    counter = counter + 1
    print("Bad_increment:", counter)

def good_increment():
    global counter
    counter = counter + 1
    print("Good_increment:", counter)
```

Error! The assignment makes Python treat `counter` as a local variable, but it is used before being assigned, causing an `UnboundLocalError`.

Local vs. global variables

```
counter = 0

def show_counter():
    print("Current counter:", counter)

def bad_increment():
    counter = counter + 1
    print("Bad_increment:", counter)

def good_increment():
    global counter
    counter = counter + 1
    print("Good_increment:", counter)
```

Explicitly use the global keyword to declare that we are modifying the global variable.

Outline

- Function
- Module
- Specification
- Testing

Module

- A module is a collection of variables and functions
  - E.g. , the `math` module: `sqrt(), cos(), pi`
- To use variables and functions in a module:
  - `import` the module
  - Access functions/variables using dot operator (.)

```
<module_name>.<func_name>(<args>)
<module_name>.<var_name>
```

Module: Example

```
>>> import math
>>> math.pi
3.141592653589793
>>> p = math.sqrt(25)
5
```

Module

- Accelerate development
  - Modules enable to build complex programs quickly by reusing existing code
- Library:
  - A large collection Python built-in modules
  - Thousands of third-party modules available on PyPI

Create your own modules

- We just need to start saving our code in files

Create your own modules

uet_info.py

```
"""
UET-VNU information.
"""
# Global variables
university_name = "UET - VNU"

def get_address():
    """
    Returns the university address
    """
    return "114 Xuan Thuy”
```

Create your own modules

uet_info.py

```
"""
UET-VNU information.
"""
# Global variables
university_name = "UET - VNU"

def get_address():
    """
    Returns the university address
    """
    return "114 Xuan Thuy"
```

Module name is determined by the file name

Docstring

Comments start with #

Use your own modules

main.py

```
import uet_info
if __name__== "__main__":
    print(uet_info.university_name)
    print(uet_info.get_address())
```

Interactive mode

```
>>> import uet_info
>>> uet_info.university_name
UET - VNU
```

Outline

- Function
- Module
- Specification
- Testing

Specification

- Often write as docstrings
- Specification specifies:
  - How to use the function
  - Not how to implement it
- Writing specifications requires an implementer to imagine being a user

```
math.sqrt(x)
```

Return the square root of *x*.

This is a specification
What the function evaluates to

Specification

- Specification of a function often contains:
  - Purpose: what it does (not how)
  - Parameters and types (+ preconditions)
  - Returns and types
  - Side effects (if any)
  - Exceptions (when raised)
  - Examples/tests

Docstring: Also part of function definition

```
def inches_to_feet(inches):
    """
    Returns inches converted to feet
    Parameter inches: the number to convert
    """
    return inches/12
```

Docstring: in tripple quotes; becomes the help() information for the function

Example

```
import math

def circle_area(radius):
    """
    Returns the area of a circle with the given
radius.

    Parameters:
        radius (float): The radius of the circle,
        must be >= 0.

    Returns:
        float: The area of the circle.
    """
    return math.pi * radius ** 2
```

Content

- Function
- Module
- Specification
- Testing

Why testing?

- Ensure the function behaves as expected
- Helps detect errors early and improve reliability
- Make maintenance and future changes safer

How to write a test

```
assert condition, "Optional error message"
```

- Condition: the expression you expect to be true
  - If the condition is True, the program continues normally
  - If the condition is False, Python raises an `AssertionError`
- Optional error message: a string to display if the assertion fails

Example

```
x = 10
assert x > 0, "x must be positive"
```

The condition is True, the program continues normally.
→ Test passed

Example

```
x = -1
assert x > 0, "x must be positive"
```

The condition is False, an error occurs.
→ Test failed

```
    assert x > 0, "x must be positive"
AssertionError: x must be positive
```

Test cases can come first

- Once you have the function specification, you can write test cases
- You don’t need an implemented function body!

Test-driven development

- Write specification
  - Describe what the function should do (inputs, outputs, edge cases)
- Create tests
  - Define expected inputs and outputs before coding
- Implement the function
- Run tests
- Refine the code
  - Fix errors until all tests succeed

What to do when a test case fails

- Don’t panic
- Don’t immediately start editing code
- Don’t avert your eyes from the failure

Instead…

Read the ERROR MESSAGE

Example: Specification

- Write a function `fib(n)` that return nth Fibonacci number, where:
  - `fib(0) = 0`
  - `fib(1) = 1`
  - `fib(2) = fib(1) + fib(0) = 1`
  - `fib(3) = fib(2) + fib(1) = 2`
  - `fib(n) = fib(n-1) + fib(n-2)`

Example: Write tests before writing code

```
# test_fib.py
# fibonacci is the file implementing fib(n) function
from fibonacci import fib

def test_first_two():
    assert fib(0) == 0
    assert fib(1) == 1

def test_small_numbers():
    assert fib(2) == 1
    assert fib(3) == 2
    assert fib(4) == 3

def test_large_number():
    assert fib(10) == 55
```

Example: Write tests before writing code

```
# test_fib.py
# ....
if __name__ == '__main__':
    test_first_two()
    test_small_numbers()
    test_large_number()
```

Example: Writing first version of `fib`

```
# fibonacci.py
def fib(n):
    """
    Returns the n-th Fibonacci number.

    Parameters:
        n (int): The position in the Fibonacci sequence.

    Returns:
        int: The n-th Fibonacci number.
    """
    return 0
```

Now, executing `test_fib.py`, an `AssertionError` occurs because the current implementation of `fib(n)` always returns 0.

Example: Refine function fib

```
# fibonacci.py
def fib(n):
    """
    Returns the n-th Fibonacci number.

    Parameters:
        n (int): The position in the Fibonacci sequence.

    Returns:
        int: The n-th Fibonacci number.
    """
    return 0
```

Your task: Refine the function `fib(n)` so that it passes all the tests in `test_fib.py` file

Summary

- Function and Module
  - Organize code into reusable blocks
- Specification
  - Describe what a function does
  - Written as docstrings
- Program testing
  - Use assert to verify expected behavior
  - TDD: write specs → write tests → implement → run tests → refine