
FUNctions, FUNctions,
FUNctions
Today is a Paper + Pencil or Tablet + Pencil day...
please keep laptops stowed away!
COMP110 - CL02
2024/01/18

Announcements

- EX00 Due Tonight - 80% have completed, great work!
- EX00 Post Reflections - Release Today, Due Sunday
- LS04 Environment Diargram Practice - Release Tomorrow, Due Monday
- EX01 Cottage Tea Party Planner - Release Sunday, Due Monday 1/29

The Anatomy of a Function Definition

```
def name_of_function(parameter: type) -> returnType:
    """Docstring description of function for people"""
    return expression_of_type_returnType
```

Function Definition *Signature*

```
def name_of_function(parameter: type) -> returnType:
    """Docstring description of function for people"""
    return expression_of_type_returnType
```

The **signature** of a function definition specifies how you and others will make use of the function from elsewhere in a program:

What is its **name**?

What input **parameter(s) type(s)** does it need? *(Think: ingredients...)*

What **type of return value** will calling it result in? *(Think: biscuits)*

Function Definition *Body* or *Implementation*

```
def name_of_function(parameter: type) -> returnType:
    """Docstring description of function for people"""
    return expression_of_type_returnType
```

The **body** or implementation a function definition specifies the subprogram, or set of steps, which will be carried out every time a function calls the definition:

Each statement in the body is **indented** by one-level to visually denote it.

The **Docstring** describes the purpose and, often, usage of a function *for people*

The function body then contains one-or-more **statements**. For now, our definitions will be simple, one-statement functions.

Return statements are special and written inside of function definitions, when a function definition is called, a return statement indicates "**stop following this function right here and send my caller the result** of evaluating this return expression!"

Fill in the Blank to Complete the Missing Expression

You are planning a garden tea party with your woodland friends and want to hang string lights around the perimeter of your porch. How long of a strand of string lights will you need?

```
def perimeter(length: float, width: float) -> float:
    """Calculate the perimeter of a rectangle"""
    return _____________________________________
```

This is an example Function Call Expression that calls the **`perimeter`** function definition above.
What value and type will this expression evaluate to?

```
perimeter(length=10.0, width=8.0)
```

Identifying... write down at least one line number for each...

```
1    """A simple program with a function call."""
2
3
4    def perimeter(length: float, width: float) -> float:
5        """Calculates the perimeter of a rectangle."""
6        return 2.0 * length + 2.0 * width
7
8
9    print(perimeter(length=10.0, width=8.0))
```

1. Docstring
2. Function Call(s)
3. Return Statement
4. Function Definition
5. Usage of a Parameter's Name in an Expression

Tracing Programs by Hand

Introduction to Environment Diagrams

- Working through the evaluation of a program depends on many interrelated values.
- As any non-trivial program is evaluated, what needs to be kept track of includes:
  1. The current line of code, or expression within a line, being evaluated
  2. The trail of function calls that led to the current line and "frame of execution"
  3. The names of parameters/variables and a map of the values they are bound to
  4. and more!
- As humans this quickly becomes more information than we can hold in our heads.

  Good news: Environment diagrams will help you keep track of it all on paper!

Environment Diagrams

- A program's runtime *environment* is the mapping of *names* in your program to their *locations* in memory.
- A program's *state* is made up of the *values stored* in those locations.
- You can use *environment diagrams* to visually keep track of both the *environment* and its *state*.
- Additionally, *environment diagrams* will help you keep track of how function calls are processed.

```python
"""A simple program with a function call."""


def perimeter(length: float, width: float) -> float:
    """Calculates the perimeter of a rectangle."""
    return 2.0 * length + 2.0 * width


print(perimeter(length=10.0, width=8.0))
```

```python
"""A program with a *two* function calls."""


def perimeter(length: float, width: float) -> float:
    """Calculates the perimeter of a rectangle."""
    return 2.0 * length + 2.0 * width


def square_perimeter(side: float) -> float:
    """Calculates the perimeter of a square."""
    return perimeter(length=side, width=side)


print(square_perimeter(side=4.0))
```

The `return` Statement vs. calls to `print`

- **The return statement** is ***for your computer*** to send a result back to the function call's bookmark *within your program.*
  - A bookmark is dropped when you *call* a function with a return type. When that function's body reaches a *return statement,* the returned value *replaces* the function call and the program continues on.
- **Printing *is for humans* to see**. To share some data with the user of the program you must *output* it in some way.
- If you have a function f that returns some value, you can print the value it returns by:
  - 1. Printing its return value directly **print(f())**, or
  - 2. (Later in the course) By storing the returned value in a variable and *later* printing the variable.

Consider the following function definition.
First: identify its *name, parameter(s), return type.*
Then: what does the function call expression evaluate to?

```
def mystery(message: int) -> str:
    """Hmmm....."""
    return message + "!"
    return message + "?"
```

Example Function Call Expression that calls the **`mystery`** function definition above.
What value and type will it evaluate to?

```
mystery(message="Fox")
```