Week 3

Chapter 1

Outline

- Function definition (basic)
- Formal and Actual Parameters
- Local Variables, Scope
- Name Resolution
- Positional and Keyword Arguments
- Default Parameter Values
- Name Resolution
- Modules
- Module Execution
- Python Coding Style

Chapter 2

Function definition (basic)

```
def <function name> ([<formal parameters>]):
    [<docstring>]
    <function body>
```

> ℹ Note
>
> Function **definitions** establish a **binding** between `<function name>` and un-evaluated `<function body>`. For the function to be able to participate in a function **call**, the body must be evaluated using the existing bindings of the **actual parameters** passed to the function during the call.

While Python functions may seem analogous to math functions, the analogy is somewhat tenuous: e.g., the former can have **side-effects** (like `print`ing something or modifying *mutable* objects)

Chapter 3

Formal Parameters Versus Actual Parameters

Formal parameters are **place-holders**: in the body of the function, they stand in for objects that are **specified** when the function is called!

The objects passed to the function call are called **actual parameters**. In the most common usage, actual parameters are passed **positionally**, i.e., the values are assigned to the formals in left-to-right order before the body is executed.

Informally, both formal and actual parameters are often referred to as **arguments** but there is a distinction: formal parameters have no values.

The process of encapsulating the function body in the definition is called **lambda abstraction**.

Chapter 4

Examples

1. Use lambda abstraction to create a general purpose function called `symmetric_in` that takes two strings as parameters and returns `True` if and only if one of the strings is contained in the other.
2. Write a function that takes a string as argument and checks whether it is a **palindrome**, i.e., reads the same forwards and backwards.

Chapter 5

Local Namespace of a Function

A function's **local** namespace:

- any formal parameter, or
- any variable name used on the **left-hand-side** of an assignment statement within the function body.

All other names referenced in the body must be **resolved** at the time of call, i.e. an appropriate binding must be found for it.

Chapter 6

The LEGB rule for name resolution

One of the most common errors that occur in Python programs is a **`NameError`**! This happens when Python's rule for resolving a name fails.

The rule is based on **lexical scope**: the **nested** structure of the **definition blocks** within a program determines how names are resolved.

In order, **LEGB**

- **L**ocal scope: current definition
- **E**nclosing scope: within any (strictly) enclosing definition
- **G**lobal scope: the top-level namespace (i.e. bound in the module)
- **B**uiltin scope: a builtin object definition

Qualified names, i.e., names with the dot notation, are resolved by looking at the sequence of namespaces obtained from the dots.

6.1 Example

```
import math
def f(x):
    print(f"Outer f's locals: {locals()}")
    print(f"Outer f's globals: {globals()}")
    return x+y

def g(x, z):
    z = 10
    def f(x, z):
        print(f"Enclosing f's locals: {locals()}")
```

```python
        print(f"Enclosing f's globals: {globals()}")
        return x**2 + y**2 + z
    print(f"Outer g's locals: {locals()}")
    print(f"Outer g's globals: {globals()}")
    return f(x, y) % z

y = 19
print(math.pi)
print(f(30))
print(g(y, 3))
```

Chapter 7

Keyword Arguments

- Arguments within the call that of the form

```
<formal>=<object>
```

> ! Important
>
> The same call can have both positional and keyword arguments. However, all keyword arguments must come **after** any positional ones (which occur in the order given by the definition). Keyword arguments do not need to be in order!

7.1 Default Parameter Values

Usually, meant to be used with keyword arguments: the default value for that argument is specified at **definition time**, and any variation at the time of call is supplied as a keyword argument referencing that parameter.

7.2 Variable number of arguments

Function definitions allow for any number of positional arguments (indicated by convention as a `*args` parameter) and any number of keyword arguments (indicated by convention as a `**kwargs` parameter).

We will study these later after we've had a chance to understand **tuple** and **dictionary** datatypes.

Chapter 8

Examples

- Check the documentation of the `pow` builtin function and see how it can be called in various ways by combining positional and keyword arguments.
- Repeat the exercise for the documentation of the `abs` builtin function and the `math.isclose` function.

Chapter 9

Modules

Python source code files (with the extension `.py`) are called **modules**.

- `import` statement allows access to the **global** namespace of the imported module
- whether a module can be imported *depends on the `PYTHONPATH` environment variable (we will study this later). For now, you should ensure that any user-defined modules are in the same folder as the program importing them.
- `import <module>` will import all global names within `<module>`: the bindings are referenced, e.g., as `<module>.<name>`
- `from <module> import <name>` allows unquilified use of `<name>`

⚠ Warning

Although allowed, you should avoid using `from <module> import *`. It can be a source of ambiguity and consequent errors in name resolution!

- `from <module> import <name> as <alias>` or `import <module> as <alias>` are common ways of *abbreviating* long names (or **sub-packages** which we will come across later)
- a module is imported only once per interpreter session

Chapter 10

Execution of a Module

Execution consists of evaluation of the definitions and the statements in the module.

Two ways in which a module can be used:

- as a source for definitions to be used in other modules (i.e., like a **library**)
- as a stand-alone program (or **script**) to be executed.

In this latter form of use, a **runtime stack** keeps track of function calls!

- the **main** frame (containing global namespace definitions) is at the bottom

Every function call results in the activation of a new frame that keeps track of the local namespace of the function.

- new frame is **pushed** on top of the **calling** program component's frame
- control flows to the function body after parameter bindings are performed per the call's arguments
- when the function `return`s successfully, its frame is **popped** from the stack
- control returns to the point of execution just after the call in the calling program's component

Chapter 11

Python Coding Style

A series of Python Enhancement Proposals (**PEP**s) have served as design documents describing new additions to the language as it evolved, including **best practices for coding style**.

- PEP 8, the style guide for Python code, and PEP 257, the docstring convention guide, form the basis for most best practices.
- Variations in docstrings and project- or company-specific guidelines usually try to stay close to these PEP conventions.

We will follow the PEPs fairly closely as well:

- variable names (including module names and function names) in **snake-case**; they should begin with lowercase letters
- avoid short variable names: the only place where they may be reasonable is in toy code (for demo purposes), or as index variables when multiple such variables are needed.
- **constant** names should be in uppercase
- we will use capitalized names in snake-case for **classes** (later + there are other style conventions associated with classes)
- follow Google-style for docstrings
- start using `pylint` or `flake8` packages to check your modules for style violations! This is called **linting**.