Programming Fundamentals 1

Lesson 3

Pierre Kelsen

Department of Computer Science (DCS - MNO)

2023

Outline

Towards Functions

Code Reuse

- In lesson 2 we saw a program for the Number Search Problem
- Suppose we want to write another program (let us call it Program 2) that needs to solve this problem multiple times
- This is an example scenario for code reuse
- We could simply copy and paste the code for the number search problem into those places where we need to solve this problem

Towards Functions

Structure of Program 2

- Program 2 would look like this:

```
# Program 2
# some statements
# code for Number Search
# some more statements
# code for Number Search
# even more statements
# code for Number Search
# yet more statements
```

Towards Functions

Problems with Copy and Paste

Question

What are drawbacks of this approach to code reuse?

Answer

Any modification to the method used for number search will entail changes in multiple places

Discussion

Why is this really a problem? Imagine real world scenarios.

Towards Functions

Programming Advice

Golden Rule of Programming

Say each thing only once!

Discussion

Why does the "copy and paste" approach to code reuse violate this rule?

Towards Functions

How to Reuse

Question

So how can we reuse code without reproducing it?

Answer

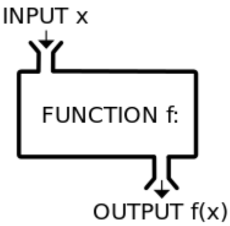
By packaging the code as a function.

Towards Functions

Functions in Mathematics

Definition (function)[1]

A function is a process or a relation that associates each element x of a set X to a single element y of another set Y (possibly the same set). We may call x the argument or input of the function and y the value or output of the function.



[1] https://en.wikipedia.org/wiki/Function_(mathematics)

Towards Functions

From Programs to Functions

- Suppose that a program solves a given problem
- We can then view the program as realizing a (mathematical) function

Question

What is missing for a program to be a full-fledged function?

Answer

We need to clearly identify inputs and outputs.

Functions in Python

Example of a Function in Python

- The following Python code computes the minimum of two numbers a and b

```
def myMin(x, y):
# x and y are the "parameters" of the function
# they constitute the "input"
    if x<y:
        return x
    else:
        return y
# output is either x or y
```

- Note: there is a builtin function `min` hence we use a different name

Functions in Python

General Form

- A Python function definition looks as follows:

```
def <name>(<parameters>):
    <body>
```

- $<$ *name* $>$ stands for the name of the function
- $<$ *parameters* $>$ stands for a comma separated list of names that constitute the input to the function
- $<$ *body* $>$ stands for a sequence of (indented) Python statements
- In previous example: $<$ *name* $>$ = myMin, $<$ *parameters* $>$ = (x, y) and $<$ *body* $>$ is composed of an `if-else` statement

Functions in Python

Function Call

- To call a function, we use the syntax: `<name>(<arguments>)`
- Arguments are expressions (that have a value)

Example

myMin(3, 4) calls function myMin with arguments 3 and 4
Parameter x will be bound to 3, and parameter y will be bound to 4

Functions in Python

Return value

- Function calls are *expressions*, meaning:
  - Like all expressions, function calls have a value
- The value is that returned by the function
- So what is value of myMin(3,4)?
- To obtain the value, execute the body with x=3 and y=4
- The value 3 is returned since $3 < 4$

Functions in Python

Function Call Execution

- To execute a function call:
  - the expressions of the arguments are evaluated, and the parameters are bound to these values
  - the point of execution moves from the point of the function call to the first statement in the body
  - the statements in the body are executed until a `return` is encountered, or there are no more statements to execute
  - in the first case the value of the function call is the value of the expression following the `return`
  - in the second case the value is *None*
  - the point of execution is transferred back to the place of the call where it continues executing the current statement

Functions in Python

Positional Parameter Binding

- In the above examples parameters were bound to arguments via their position, i.e., 1st parameter $\rightarrow$ 1st argument, 2nd parameter $\rightarrow$ 2nd argument,...
- Python also supports keyword arguments: these arguments are bound to the parameters via the parameter name
- A keyword argument must not (i.e., is not allowed to) be followed by a non-keyword argument.

Example

The call `myMin(3,4)` can be replaced by `myMin(x=3,y=4)` or even `myMin(y=4,x=3)`. These calls are all equivalent.
The call `myMin(x=3, 4)` is not allowed (see last item above).

Functions in Python

Optional Parameters

- Parameters may supply a default value using the syntax:
  - `<param> = <defaultValue>`
- A corresponding argument is optional: if it is not given, the parameter binds to the default value

Example

In the function definition def sortList(numberList, ascending = True): the second parameter has a default value.
The call `sortList(list)` is equivalent to `sortList(list, True)`

Functions in Python

Optional Parameters - Continued

- The following restriction exists for the definition of default values:
  - All parameters with default values must follow all parameters without default values.
- Keyword arguments are commonly used in conjunction with default parameter values.
- In general use of keyword arguments may improve the readability of the code
  - E.g.,the call `sortList(l, ascending=False)` documents the purpose of the second parameter.

Functions in Python

To Return or Not to Return

- The functions above all contain return statements followed by an expression
- A function may also not contain a return statement or a return statement without an expression
- Example

```
def meaningOfLife():
    print('Not sure what it is')
```

- This is a valid function definition

Functions in Python

None as a Return Value

Question

In the example on the previous slide, does the function have a return value?

Answer

Yes, it has! In those case the function returns `None`.

Functions in Python

None

- None is a special value in Python
- Since all values are objects, it is an object.
- A function that contains no return statement (or a return not followed by an expression) returns `None`

Question

What is the output when we run the following piece of code? → DEMO

```
def meaningOfLife():
    print('Not sure what it is')
print(meaningOfLife()==None)
```

Functions in Python

What is None? - continued

Question

What is the type of `None`? How can you find this out?

Answer

We can invoke the builtin `type` function. DEMO.
The type of `None` is NoneType. `None` is the only value of NoneType.

Functions in Python

Local Variable

- A local variable in a function is a variable on the left side of an assignment statement within a function

Example

In the following function, y is a local variable

```
def f(x):
    print(x)
    y=1
    print(x+y)
```

Functions in Python

Global versus Local Variables

Example

Consider the following program:

```
def f(x):
    y=1
    print(y)
y=3
f(y)
print(y)
```

Question

What is the output?

Answer

1
3

Functions in Python

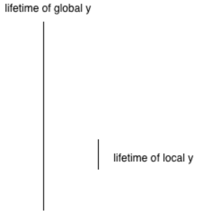
Global versus Local Variables - Continued

- The assignment `y = 3`, being outside any function, defines a global variable
- The assignment `y = 1`, being inside function f, defines a local variable
- These are two different variables

Functions in Python

Lifetime of Variables

- Global variables exist as long as the program has not terminated
- Local variables are created every time the function is executed and removed when the function ends
- When the example program executes we have the following situation



Functions in Python

Meaning of Variable Names

- Suppose a statement is executed that is referring to y.
- How do we know which variable we are referring to?
- Answer: during the lifetime of the local variable any reference to y refers to the local variable, any other time it refers to the global variable.

Functions in Python

Referencing Global Variables

- Discuss the following program

```
def f():
    print(y)
y=3
f()
print(y)
```

- There is a single global variable y.
- The global variable is referenced inside the function.
- Thus the ouput is:
  - 3
  - 3

Functions in Python

Referencing Global Variables - Continued

Question

Is it a good idea to access global variables from inside a function?

Answer

No, it's not a good idea because ...
... it makes programs more difficult to read
... it makes functions more difficult to reuse (why?)

Functions in Python

Programming Advice

Advice

It is generally not a good idea to reference global variables from a function. If the function needs to access information at the global level, this information should be passed via parameters.

Functions in Python

Applying the Advice

- So let us apply the advice to the previous program

```
# BEFORE
def f():
    print(y)
y=3
f()
print(y)
```

```
# AFTER
def f(z):
    print(z)
y=3
f(y)
print(y)
```

Functions in Python

Alternative

- Note that we could have done this:

```
# BEFORE
def f():
    print(y)
y=3
f()
print(y)
```

```
# AFTER
def f(y):
    print(y)
y=3
f(y)
print(y)
```

- Would the final program behave in the same way?
- Yes, it would as parameters are treated as local variables: during the function execution y refers to the parameter.

Functions in Python

Programming Advice

Advice

It is recommended to give parameters names different from those of global variables to avoid confusion and enhance readability.

Functions in Python

A Simple Function

- Consider the following function:

```
def sumOfSquares(n):
    """ Assumes n is an integer with n>=0
        Returns the sum of squares of numbers from 0 to n """
    s = 0
    for i in range(n+1):
        s += i * i
    return s
```

Question

What do you notice about this function?

Functions in Python

Docstrings

- The function `sumOfSquares` of squares contains a special comment (multi-line!) enclosed by triple quotes (watch the indent!)

```
""" Assumes n is an integer with n>=0
    Returns the sum of squares of numbers from 0 to n """
```

- Such a comment is called a docstring in Python

Question

What is the purpose of this particular comment?

Answer

To define the contract between the user of the function and the implementer (programmer). We also talk about the specification of a function

Functions in Python

Functions and Contracts

- With each function we can associate a contract/specification consisting of two parts:
  - Assumptions (also called preconditions): conditions that must be met by users or clients of the function
  - Guarantees (also called postconditions) that must be met by the function.
- Provided that the assumptions are satisfied, the guarantees are required to hold

Discussion

Can you think of analogies in the real world?

Functions in Python

Assumptions

- Assumptions typically define constraints on the parameters of the function
  - the type of the parameters
  - other (boolean) conditions on the parameters

Question

What happens when an assumption is violated?

Answer

Errors or other unpredictable behavior may occur.

Functions in Python

Violating an Assumption

- Let us explore what happens in the example when the assumption is violated → DEMO

```
def sumOfSquares(n):
    """ Assumes n is an integer with n>=0
        Returns the sum of squares of numbers from 0 to n """
    s = 0
    for i in range(n+1):
        s += i*i
    return s
```

Question

Can you explain the observed behavior?

Functions in Python

Benefits of contracts

Question

Who benefits from the use of contracts?

Answer

The programmer because he knows what he must implement.
The client of the function because ideally the contract is enough to understand what the function does

Functions in Python

Information Hiding

- The use of contracts is an example of information hiding
- The client does not need to read the detailed implementation to understand what the function does

Question

Why is this use of information hiding useful?

Answer

It shields the client from the complexity of the implementation.

Functions in Python
Viewing docstrings

- Python presents a convenient way to view docstrings
- Use syntax "help(sumOfSquares)" to view contents of docstring
- Typing "sumOfSquares(" in shell or editor will display list of parameters and the first few lines of the doctring
- DEMO

Modules

Software Complexity

- We have presented functions as a way to facilitate reuse by encapsulating commonly used functionalities
- We can also view it as being helpful to tame (or control) the complexity of a program
- The structural complexity of a program is also known as *software complexity*.

Discussion

Why is software complexity a problem?

Modules

Definition of Modules

- All the programs we have seen so far are contained in a single Python file (with extension .py)
- When programs get large, it is natural to try to divide them up into different files to reduce software complexity
- Each such file is called a *module*
- A module contains a collection of related Python definitions and statements

Modules

Benefits of Modules

- Modules help to reduce the software complexity of a program
- Other uses are:
  - for software development: dividing up programming tasks
  - for software testing: allows different functionalities to be tested separately
  - for software maintenance: facilitates modification of specific program functionalities

Modules

Module Example

- Example of a module circle.py

```
pi = 3.14159
def area(radius):
    return pi*(radius**2)
def circumference(radius):
    return 2*pi*radius
def sphereSurface(radius):
    return 4.0*area(radius)
def sphereVolume(radius):
    return (4.0/3.0)*pi*(radius**3)
```

Modules

Modules and Namespaces

- Each module provides its own namespace
- To use a module, you can import it as follows:
  - `import <moduleName>`
- The namespace provides a context for the names in a module
- Two different modules can be imported having functions with the same name
- These functions are differentiated using dot notation

Modules

Modules and Namespaces - Example

Example

Suppose we have two implementations of a circle module called *circle1* and *circle2* with the same function and variable names. We could then test values of pi constants as follows:

```
import circle1, circle2
...
if circle1.pi == circle2.pi:
    print('Values of pi are the same')
else:
    print('Values of pi are different')
```

Modules

Other Forms of Import

Example

We can also import using the following syntax:

```
from <moduleName> import <something>
```

- Here <something> can either be
  - a list of identifiers, e.g., func1, func2
  - a single renamed identifier, e.g.,
    - `from <moduleName> import f1 as func`
  - an asterisk, as in:
    - `from circle import *`
    - imports all names from circle module

Modules

Other Forms of Import (2)

- There is a fundamental difference between this second version of import and the earlier one:
  - For this version the namespace of the imported module *becomes part of* the importing module
  - Identifiers can thus be used without the dot notation

Caveat[2]

If there is a name clash between an imported identifier and a local identifier, the local identifier will be *masked* (or hidden).
The form with the asterisk makes name clashes more likely.

[2] "caveat" is a synonym for "warning"

Modules

Programming Advice

Advice

With the `from-import` mechanism, name clashes become possible. Because of this, `import <moduleName>` is the preferred form of import in Python.

Modules

Main Module

- When running a Python program file, this module becomes the main module
- The *current directory* is the directory containing the main module
- The namespace for the main module is the *global namespace*
- The namespace is reset every time the interpreter is started

Modules

Viewing the Namespace

- We can view the items in the namespace by executing the dir() function
- DEMO: test this after restarting the shell
- DEMO: redo this test after running an external file

Modules

Loading Modules

- Each imported module needs to be located and loaded
- Python first searches for modules in the current directory
- It then searches the directories in sys.path variable
- To view this path either execute the following code ...

```
import sys
sys.path
```

- ... or check the "Path browser" in the File menu of Idle
- DEMO

Modules

Information Hiding with Modules

- All identifiers in an imported Python module are public
- Sometimes one wants to restrict access to an item (Why?)
- Python only allows to provide a "hint" for this
  - if a name starts with two undescores (__), it is intended to be private
  - private entities should not be accessed
- if a module is imported using `from <moduleName> import *`, then private identifiers are not imported

Modules

Three Namespaces

- When a program executes, there are up to three name spaces active
  - Global namespace = namespace of currently executing module
  - built-in namespace = namespace of builtin functions and constants
  - local namespace: namespace of currently executing function
- An identifier in a namespace can mask an identifier in another namespace

Modules

Masking of Identifiers

Example

When a function f executes with a local variable y, and a variable with that name exists in the global namespace, the local variable masks the global one (see slide 25) When a module is imported using ""`from <moduleName> import *`" and that module defines a function of the same name as a builtin function, the imported name masks the builtin function.