1811/2807/7001ICT
Programming Principles

School of Information and Communication Technology
Griffith University

Trimester 1, 2024

16 Defining Functions

In this section we learn how to define our own functions.

16.1 Why create functions?

In all programming languages we encapsulate statements into small packages called variously “subroutines”, “procedures”, “functions”, or “methods”.

We do this for three main reasons:

1. for reuse;
2. to minimize the number of defects (bugs) by simplifying a program; and
3. as another way to control the flow of execution in a program.

16.1.1 Reuse

If we can identify a group of statements that carry out an action that we want to perform more than once in a program, it is better to package those statements in a function. The overall program will be smaller and easier to maintain.

We may be able to reuse a function in another program.

Cohesion (how well the parts stick together) is an important property of functions.

A function with a single well-defined purpose is more likely to be reusable.

16.1.2 Minimizing defects

As jobs get more complex, the people doing them make more errors.

The more statements and variables you have to take into account at the same time, the more likely you are to make errors.

To solve big problems we use the divide and conquer strategy.

Large programs broken down into many functions will be easier to build with fewer defects, and also easier to understand later as the program is maintained.

Functions should rarely contain more than about 20 statements.

16.1.3 Flow control

The flow of control of a program can be deviated from the normal sequence by using loops and selections, as we have already been doing.

Calling a function also alters the flow of control.

Functions can call themselves (*recursion*). This is a way to get repetition in a program without writing loops.

16.2 Defining functions

We define our own functions with the `def` statement.

Template:

```
def  function-name(argument(s)):
   statements(s)
```

Because Python is an interpreted language, that is executed as it is read, the definition of a function must come before any calls to it.

Example 1:

```
# script: function1.py
# Declare and call a function:
#   with no arguments or return value
```

```
def printBoo():
   print("Boo!")

printBoo()
printBoo()
printBoo()
```

```
$ python3 function1.py
Boo!
Boo!
Boo!
$
```

16.2.1 Docstrings

A convention for documenting Python code is to include a multiline string as the first statement in a function, module, or class.

The first sentence should briefly describe the purpose.

After a blank line, further details may be added.

Docstrings are used by many python tools, including the `help()` function in the read-eval-print loop ().

16.2.2 Simple arguments

This example has a function that takes an argument and prints it vertically.

First, whatever type the argument is, it must be converted to a string.

```
# script: function2.py
# Declare and call a function:
#   with an argument, but no return value

def printVertical(x):
   """Print x vertically."""
   for c in str(x):
      print(c)

printVertical("Boo!")
printVertical(42)
```

```
$ python3 function2.py
B
o
o
!
4
2
$
```

16.2.3 Returning values

The `return` statement exits a function, optionally returning a value.

```
# script: function3.py
# Declare and call a function:
#   with multiple arguments, and a return value

def finalBalance(p, r, n):
   """Returns a final balance with compound interest.

      p in the principle (initial balance)
      r is the interest rate per term
      n is the number of terms"""
   return p * (1.0 + r) ** n
```

```
initBal = float(input("Initial balance: "))
annRatePct = float(input("Annual interest rate (%): "))
months =  int(input("Number on months: "))
finalBal = finalBalance(initBal,
             annRatePct / 100.0 / 12.0, months)
print("Final balance = ${:.2f}".format(finalBal))
```

```
$ python3 function3.py
Initial balance: 100.00
Annual interest rate (%): 10.0
Number on months: 24
Final balance = 122.04
$
```

16.2.4 Keyworded arguments with default values

Function arguments may have default value.

Such arguments are then optional in the call, but must be identified by name, and follow the other arguments.

```
# script: function4.py
# Declare and call a function:
#   with multiple arguments, some keyworded

def printBox(width, height, empty = False, framed = False):
   """Print a patterned  box of size width x height
      characters."""
   for i in range(height):
      for j in range(width):
         if framed and \
```

```
            (i == 0 and (j == 0 or j == width - 1) or \
            i == height - 1 and (j == 0 or j == width - 1)):
            print("+", end = "")
         elif framed and (i == 0 or i == height - 1):
            print("-", end = "")
         elif framed and (j == 0 or j == width - 1):
            print("|", end = "")
         elif not empty and (i + j) % 2 == 0:
            print("X", end = "")
         elif not empty:
            print("O", end = "")
         else:
            print(" ", end = "")
      print()

printBox(5, 3)
printBox(10, 4, framed = True)
```

```
printBox(10, 4, framed = True, empty = True)
```

```
$ python3 function4.py
XOXOX
OXOXO
XOXOX
+--------+
|XOXOXOXO|
|OXOXOXOX|
+--------+
+--------+
|        |
|        |
+--------+
$
```

16.3 Scope, global and local

> *Definition:* The *scope* of a variable is the region of the program in which the variable may be accessed. A variable is visible to the statements that try to use it, if it is *in scope*.

When we introduce functions, we introduce the concept of scope, because now we have two kinds of variables, with two types of scope, *local* and *global*.

- Global variables are those created by statements outside of any function.
- Local variables are those created inside a function.

16.3.1 Local variables

This program has a function that defines the variable `x` inside a function.

Then in the main statements below, it tries to print it after calling the function.

```
# script: scope1.py

def f():
   x = 2
   print("a x =", x)

y = 1
f()
print("b y =", y)
print("c x =", x)
```

`a`, `b` and `c` are labels that help us to know which `print` call prints what.

```
$ python3 scope1.py
a x = 2
b y = 1
Traceback (most recent call last):
  File "scope1.py", line 10, in <module>
    print("c x =", x)
NameError: name 'x' is not defined
$
```

Inside the function we can print `x`, but outside the function, we can't, because it is not in scope.

Local variables are only visible inside the function they are defined in.

16.3.2 Global variables

In this program we try to print the global variable y from inside the function.

```
# script: scope2.py

def f():
   x = 2
   print("a x =", x)
   print("b y =", y)

y = 1
f()
print("c y =", y)
```

```
$ python3 scope2.py
a x = 2
b y = 1
c y = 1
$
```

Functions can normally see the values of global variables, but they can not normally assign new values to them.

Make sure the global variable has been created before the function is called.

```
# script: scope3.py

def f():
   x = 2
   print("a x =", x)
   print("b y =", y)

f()
y = 1
print("c y =", y)
```

```
$ python3 scope3.py
a x = 2
Traceback (most recent call last):
  File "scope3y.py", line 8, in <module>
    f()
  File "scope3y.py", line 6, in f
    print("b y =", y)
NameError: name 'y' is not defined
$
```

16.3.3 Lifetimes of variables

Local variables don't just go out of scope when their function exits.

They cease to exist, and the Python interpreter can reuse the memory they used to use.

Global variables exist until the program ends (unless deleted with `del`).

So the lifetime of a variable depends upon its scope.

16.3.4 Parameters and parameter passing

Function parameters behave much like local variables, with the exception that when the function is called, values are passed to them from the function call.

This program's function has a parameter `x` that is passed a copy of global variable `y`.

```
# script: scope4.py

def f(x):
   print("b x =", x)
   x += 1
   print("c x =", x)

y = 1
print("a y =", y)
f(y)
print("d y =", y)
```

```
$ python3 scope4.py
a y = 1
b x = 1
c x = 2
d y = 1
$
```

Changing `x` with an assignment inside the function does not change the global variable `y`.

Because the values bound to these variables are immutable `int` objects, there is nothing that can be done to or with `x` that will affect `y`.

This is *not* the case for mutable objects.

Let's rerun the program with diagrams.

First the global variable `y` is created and printed.

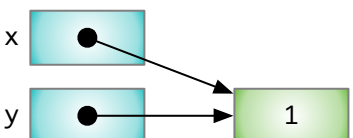
statements executed | *diagram* | *output*

```
y = 1
print("a y =", y)
```


```
a y = 1
```

Then the function `f` is called, passing a copy of `y` (the reference!) to parameter `x`. Both point to the same `int` object.

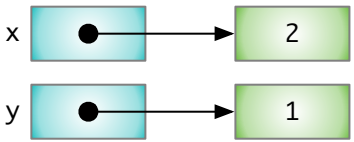
```
f(y)

def f(x):
   print("b x =", x)
```




```
b x = 1
```

Then `x` is assigned a new value, so it points to another object. `y` is unchanged.

```
x += 1
print("c x =", x)
```




```
c x = 2
```

The function exits. Parameter `x` no longer exists.

```
print("d y =", y)
```


```
d y = 1
```

16.3.5 Eclipsing globals

What happens when a global variable and a parameter or a local variable have the same name?

In this program a local and a global variable have the same name.

```
# script: scope5.py

def f():
   x = 2
   print("b x =", x)

x = 1
print("a x =", x)
f()
print("c x =", x)
```

```
$ python3 scope5.py
a x = 1
b x = 2
c x = 1
$
```

Assigning a value to the name `x` in the function, creates a new local variable.

It does not change the global variable.

From then on, the global variable becomes invisible to the function, because the function will only see the local variable.

The global name has been *eclipsed* by the local name.

16.3.6 Don't use globals inside a function

Though we can normally see the values of global variables from inside functions there are good reasons not to use global variables inside functions, and they relate to why we use functions at all.

- Functions might be reusable.

  If we write useful function, we might want to use it in our next program.

  But if it depends on global variables, when we copy it to the new context it will not work, because the globals are not there.

- Functions are supposed to reduce the complexity of programs, minimising defects, and making programs easier to write and debug.

  This is true if we can reason about a function by itself, without worrying about the state of global variables that it uses.

  If a function relies on global variables, then other code will have *side-effects* on the behaviour of the function.

So to improve the quality of your programs:

- don't use *any* global variables inside your functions;
- use *only* parameters and return values to communicate between functions; and
- you can go further – have as *few* global variables in your program as possible.

These ideas apply in any programming language.

Pure functional programming languages, such as Haskell, do not have global variables (that can change) at all!

This *eliminates* side-effects.

Section summary

This section covered:

- the reasons we organise our code into functions, reuse, simplifying, and control flow;
- how we define functions with `def`;
- docstrings;
- simple and keyworded arguments, and returned values;
- scope and lifetimes of variables and parameters.