Unit 4

Functions and Decomposition

Learning Outcomes

- Design and implement functions to carry out a particular task.
- Begin to evaluate when it is necessary to split some work into functions.
- Locate the parts of a program where particular variables are available.
- Import Python modules and use their contents.
- Read the Python module reference for information on a module's contents.
- Use objects provided by modules in your programs.
- Catch errors in programs and handle them gracefully.

Learning Activities

- Read this unit and do the "Check-Up Questions."
- Browse through the links for this unit on the course web site.
- Read Sections 3.6–3.12, 4.8, 5.1–5.4, 5.6 in *How to Think Like a Computer Scientist*.

Topic 4.1 Defining Functions

We have already seen how several *functions* work in Python. In particular, we have used `raw_input`, `range`, `int`, and `str`. Each of these is built into Python and can be used in any Python program

```
def read_integer(prompt):
    """Read an integer from the user and return it."""
    input = raw_input(prompt)
    return int(input)

num = read_integer("Type a number: ")
print "One more is", num+1

num = read_integer("Type another: ")
print "One less is", num-1
```

Figure 4.1: A program with a function defined

A function must be given *arguments*. These are the values in parentheses that come after the name of the function. For example, in `int("321")`, the string `"321"` is the argument. Functions can have no arguments, or they can take several.

Functions that *return* values can be used as part of an expression. We saw how the `int` function works, which returns an integer. It can be used in an expression like this:

```
x = 3*int("10") + 2
```

After this statement, the variable `x` will contain the number 32. In this expression the `int` function returns the integer 10, which is then used in the calculation.

Python functions can return any type of value including strings and floating point values.

Defining your own functions

You can define your own functions as well. They are defined with a `def` block, as shown in Figure 4.1. The code inside the `def` isn't executed right away. The function is defined and then run whenever it is *called*.

In Figure 4.1, the function is named "`read_integer`" and takes one argument that we'll call `prompt`. Inside the function definition, `prompt` works like a variable. Its value is filled in with whatever argument is given when the function is called.

The next line is a triple-quoted string that describes the function. This is called a *documentation string* or *docstring*. The docstring is a special form of a comment in Python: it has no effect on the behaviour of the function. It works like a comment and will help somebody reading your code figure out what it does.

Every function you write in this course must have a meaningful docstring. It will help us understand your code more easily when we mark it. It is also a good habit to get into. When you have to come back to some of your own code after a few weeks, you'll be glad you included it.

The statements in the body of the function are what will be executed when the function is called. The `return` statement indicates the value that the function returns.

The main part of the program in Figure 4.1 makes two calls to the `read_integer` function. Here's what the program looks like when it's run:

```
Type a number: 15
One more is 16
Type another: 192
One less is 191
```

You should define functions to do tasks that you'll have to do several times. That way you'll only have to type and debug the code once and be able to use it many times. As a general rule, you should never copy-and-paste code. If you need to reuse code, put it in a function and call it as many times as necessary.

Defining functions is also useful when you are creating larger program. Even if you're only going to call a function once, it helps you break your program into smaller pieces. Writing and debugging many smaller pieces of code is much easier than working on one large one.

Calling Functions

Consider the example function in Figure 4.2. This does a calculation that is common in many algorithms. The docstring should be enough for you to figure out what it does.

Suppose we then run this statement:

```
half_mid = middle_value(4,2,6) / 2
```

```
def middle_value(a, b, c):
    """
    Return the median of the three arguments.  That is,
    return the value that would be second if they were
    sorted.
    """
    if a <= b <= c or a >= b >= c:
        return b
    elif b <= a <= c or b >= a >= c:
        return a
    else:
        return c
```

Figure 4.2: A sample function

What exactly happens when the computer "runs" this code?

1. The expression on the right of the variable assignment must be evaluated before the variable can be assigned, so that is done first. It evaluates the expression `middle_value(4,2,6) / 2`.
2. The sub-expressions on either sode of the division operator must be evaluated. The first is a call to our function. It then evaluates the expression `middle_value(4,2,6)`.
3. This requires calling the function in Figure 4.2. Now, this statement is put on hold while the function does its thing.
4. The function `middle_value` is called.
   (a) The arguments that are given in the calling code (`4,2,6`) are assigned to the local variables given in the argument list (`a,b,c`). So, the behaviour is as if we had code in the function that made these assignments: `a=4`, `b=2`, `c=6`.
   (b) The code in the function body then starts to execute. This code executes until it gets to the end of the function or a `return` statement. It this case, the `if` condition is false, so the `return b` statement is skipped. The condition of the `elif` is true (since `b <= a <= c`), so the `return a` statement executes.

   (c) The function returns the integer `4` and exits. Any code after the `return` doesn't execute, since the function has already stopped.

5. The calling code gets the return value, `4`. This is used in place of the function call. The expressions is now `4/2`.
6. The division is done. The original expression has evaluated to the integer `2`.
7. The integer `2` is assigned to the variable `half_mid`.

This outline of a function call is reasonably representative of any function call. When the call occurs, that code pauses until the function is finished and returns a value.

Functions that don't return values are similar. The only difference is that they are not part of a larger expression. They just execute and the calling code continues when they are done.

Why Use Functions?

Functions can be used to break your program into logical sections. You can take a specific task or calculation, and define a function that accomplishes that task or calculation. Breaking the logic of a program up into sections can make it much easier to build. You can create functions to handle parts of your algorithm, and assemble them in a much simpler main program.

Using functions well can make your program much easier to read. Functions should have descriptive names, like variables. The function should be named after what it does or what it returns. For example, `read_data_file`, `initial_guess`, or `run_menu`.

The function definitions themselves can be relatively small (and understandable) stretches of code. Someone trying to read the program can figure out one function at a time (aided by a good function name and the docstring). Then, they can move on to the main program that assembles these parts. This is generally much easier than reading (and writing and debugging) one long section of code in the main program.

Functions are also quite useful to prevent duplication of similar code. If you need to do similar tasks in different parts of the program, you could copy-and-paste code, and many beginning programmers do. But, what happens when you want to change or update that task? You have to hunt for that

code everywhere it occurs and fix it in every location. This is tedious and error-prone.

If the repeated task is separated into a function, then maintaining it is much easier. It only occurs in one place, so it's easy to fix. You should *never* copy-and-paste code within a program—it creates a maintainance nightmare. Functions are one tool that can be used to unify tasks.

Check-Up Questions

- Write a function `square` that takes one floating point value as its argument. It should return the square of its argument.
- Have a look at programs you've written for this course. Are there places where some work has been duplicated and could be put into a function?

Topic 4.2 Variable Scope

In Figure 4.1, the argument `prompt` is only available in the `read_integer` function. If we tried to use `prompt` outside of the function, Python would give the error

```
NameError: name 'prompt' is not defined
```

It does so because `prompt` is a *local variable* in the `read_integer` function. You could also say that the variable's scope with the `read_integer` function. Any variables that are created within a function are local to that function. That means that they can't be used outside of the function.

This is actually a very good thing. It means that when you write a function, you can use a variable like `num` without worrying that some other part of the program is already using it. The function gets an entirely separate thing named `num`, and anything named `num` in the rest of the program is undisturbed.

Have a look at the program in Figure 4.3. When it's run, it produces output like this:

```
How many lines should I print? 4
*
**
***
****
```

```
def stars(num):
    """
    Return a string containing num stars.
    This could also be done with "*" * num, but that
    doesn't demonstrate local variables.

    >>> print stars(5)
    *****
    >>> print stars(15)
    ***************
    """
    starline = ""
    for i in range(num):
        starline = starline + "*"
    return starline

num = int(raw_input("How many lines should I print? "))
for i in range(num):
    print stars(i+1)
```

Figure 4.3: A program that takes advantage of local variables

There is no confusion between the variable `num` in the function and the one in the main program. When the function uses the variable `num`, it is totally unrelated to the one in the main program. The same thing happens with `i`. It is used as the loop variable for both `for` loops. Since the function has its own version of `i`, there's no conflict and both loops do what they look like they should

So, to use the function `stars`, you don't have to worry about how it was implemented—what variables names were used and for what. All you have to know it what it does.

If a programming language doesn't have this property that variables are usually local to a particular function or other part of the program, it becomes *very* hard to write large programs. Imagine trying to write some code and having to check 20 different functions every time you introduce a new variable to make sure you're not using the same name over again.

Also notice that the docstring in Figure 4.3 is much longer. It includes two examples of what the function should do. Giving examples is a good idea because it gives you something to check when you test the function. The actual behaviour should match what you've advertised in the docstring.

There is actually a Python module called `doctest` that looks through your docstrings for things that look like examples of the function's use. It then checks them to make sure the examples match what actually happens.

Why Use Functions?

Local variables introduce another reason to use functions. Since variables in functions are separate from the variables elsewhere in the program, the code has very limited interaction with the rest of the program. This makes it much easier to debug programs that are separated with functions.

Functions take in values as arguments, and can return a value when they are done. They have no other way to interact with variables in other functions.

That means that they can be debugged separately: if a function does its job given the correct arguments, then it works. If there is a problem in other parts fo the program, we don't have to worry about other functions changing variable values because they can't. Each function can be checked for correctness on its own.

```
import time
print "Today is " + time.strftime("%B %d, %Y") + "."
```

Figure 4.4: Program that prints today's date.

Since it's much easier to work with many small chunks of code than one large one, the whole writing and debugging process becomes much easier. As a programmer, you have to create and test individual functions. Once you're reasonably sure the function is corret, you can forget about it and move on to other tasks.

Topic 4.3 Python Modules

In most programming languages, you aren't expected to do everything from scratch. Some prepackaged functions come with the language, and you can use them whenever you need to. These are generally called *libraries*. In Python, each part of the whole built-in library is called a *module*.

There are a lot of modules that come with Python—it's one of the things that experienced programmers tend to like about Python.

For example, the module `time` provides functions that help you work with times and dates. The full documentation for the `time` module can be found online. It's important that a programmer can find and interpret documentation like this. It might seem daunting at first—the documentation is written for people who know how to program—it should get easier with practice.

In the documentation, you'll find that the `time` module has a function `strftime` that can be used to output the current date and time in a particular format. The program in Figure 4.4 uses the `time` module to output the current date.

When the program runs, it produces output like this:

```
Today is December 25, 2010.
```

The first line in Figure 4.4 *imports* the `time` module. Modules in Python must be imported before they can be used. There are so many modules that if they were all imported automatically, programs would take a *long* time to

start up. This way, you can just import the modules you need at the start of the program.

In the next line, the function `strftime` is referred to as `time.strftime`. When modules are imported like this, you get to their contents by calling them *`modulename.function`*. This is done in case several modules have functions or variables with the same names.

You can also import modules so that you don't have to do this: You could just call the function as `strftime`. To do that, the module's contents are imported like this:

```
from time import *
```

This is handy if you want to use the contents of a particular module *a lot*.

How did we know that `"%B %d, %Y"` would make it output the date in this format? We read the documentation online. The `%B` gets replaced with the full name of the current month, `%d` with the day of the month, and `%Y` with the four-digit year.

There are Python modules to do all kinds of things, far too many to mention here. There is a reference to the Python libraries linked from the course web site.

We will mention a few more modules as we cover other topics in the course. You can always go to the reference and get a full list and description of their contents.

Check-Up Questions

- Have a look at the module reference for `time` and see what else is there.
- Look at the other modules available in Python. You probably won't understand what many of them do, but have a look anyway for stuff that you do recognize.

Topic 4.4 Objects

As you start writing programs, you will often have to represent data that is more complicated that a "number" or "string". There are some other types that are built into Python. There are also more complicated types that can hold collections of other information. These are called *objects*.

Most modern programming languages have the concept of objects. You can think of an "object" in a programming language like a real-world object like a DVD player.

A DVD player has some buttons you can press that will make it do various things (play *this* DVD, go to menu, display information on-screen) and it displays various information for you (playing, stopped, current time). Each of the buttons correspond to various actions the player can take. Each item that is displayed reflects some information about the current state of the player.

Objects in a programming language are similar. Objects are collections of *properties* and *methods*.

A property works like a variable. It holds some information about the object. In the DVD player example, the current position in the movie might be a property. Part way through the movie, the position might be 1:10:41. You could use the remote to change this property to 1:00:00 if you want to re-watch the last ten minutes. In Python, you can set the value of a property directly, just like a variable.

A method works like a function. It performs some operation on the object. For the DVD player, a method might be something like "play this DVD". A method might change some of the method's properties (like set the counter to 0:00:00) and perform some other actions (start the disc spinning, put the video on the screen).

A particular kind of object is called a *class*. So in the example, there is a class called "DVD Player". When you create an object in the class, it's called an *instance*. So, your DVD player is an instance of the class "DVD Player".

An instance behaves a lot like any other variable, except it contains methods and properties. So, objects are really variables that contain variables and functions of their own.

Objects in Python

Classes in Python can be created by the programmer or can come from modules. We won't be creating our own classes in this course, just using classes provided by modules.

To instantiate an object, its *constructor* is used. This is a function that builds the object and returns it. For example, in Python, the module `datetime` provides a different set of date and time manipulation functions

```
import datetime

newyr = datetime.date(2005, 01, 01)
print newyr.year                    # the year property
print newyr.strftime("%B %d, %Y")   # the strftime method
print newyr
```

Figure 4.5: Date manipulation with the `datetime` module's objects

than the `time` module we saw in Topic 4.3. The `datetime` module provides everything in classes which contain all of the functions that can work on particular kinds of date information.

The `datetime` module provides the class called "`date`" which can hold information about a day. Figure 4.5 shows an example of its use.

After the `datetime` module is imported, a `date` object is created. The constructor for a date object is `datetime.date()`—this function from the `datetime` module returns a `date` object. This object is stored in the `newyr` variable.

Now that we have an object to work with, we can start poking around at its properties (variables inside the object) and methods (functions inside the object).

In the first `print` statement, you can see that a `date` object has a property called `year`. You get to a property in an object the same way you get to a function inside a module: use the name of the object, a dot, and the name of the property. The `year` behaves like a variable, except it's living *inside* the `date` object named `newyr`.

The second `print` statement shows the use of a method. Date objects contain a method called `strftime` that works a lot like the function from the `time` module. The `strftime` method takes whatever date is stored in its `date` object and formats that the way you ask it to.

Finally, we see that a `date` object knows how to convert itself to a string if we just ask that it be printed. By default, it just uses a year-month-day format.

So, the program in Figure 4.5 produces this output:

```
2005
January 01, 2005
2005-01-01
```

The ways you can use an object depend on how the class has been defined. For example, some classes know how they can be "added" together with the + sign, but `date` doesn't:

```
>>> import datetime
>>> first = datetime.date(1989, 12, 17)
>>> print first
1989-12-17
>>> print first+7
TypeError: unsupported operand type(s) for +:
'datetime.date' and 'int'
```

So, Python doesn't know how to add the integer 7 to a date. But, it does know how to subtract dates:

```
>>> import datetime
>>> first = datetime.date(1989, 12, 17)
>>> second = datetime.date(1990, 1, 14)
>>> print second-first
28 days, 0:00:00
>>> type(second-first)
<type 'datetime.timedelta'>
```

So, something in the definition of the `date` class says that if you subtract two dates, you get a `timedelta` object. The `timedelta` class is also defined by the `datetime` module and its job is to hold on to lengths of time (the time between event A and event B).

This is where the power of objects begins to show itself: a programmer can create objects that represent any kind of information and "know" how to do many useful operations for that type of information. Particularly when writing larger programs, classes and objects become very useful when it comes to organizing the information your program needs to work with.

Object oriented programming is important in modern programming. It will be introduced in more detail in CMPT 125 and 225.

Check-Up Question

- Have a look at the module reference for `datetime`. What can you do with a `timedelta` object? What other classes are provided?

```
m_str = raw_input("Enter your height (in metres): ")

try:
    metres = float(m_str)
    feet = 39.37 * metres / 12
    print "You are " + str(feet) + " feet tall."
except:
    print "That wasn't a number."
```

Figure 4.6: Catching an exception

Topic 4.5 Handling Errors

So far, whenever we did something like ask for user input, we have assumed that it will work correctly. Consider the program in Figure 2.7, where we got the user to type their height and converted it to feet. If the user enters something that can't be converted to a float, the results are not very pretty:

```
Enter your height (in metres): tall
Traceback (most recent call last):
  File "inches0.py", line 1, in ?
    metres = float(raw_input( \
ValueError: invalid literal for float(): tall
```

This isn't very helpful for the user. It would be much better if we could give them another chance to answer or at least a useful error message.

Python lets you catch any kind of error, as Figure 4.6 shows. Here are two sample runs of that program:

```
Enter your height (in metres): tall
That wasn't a number.

Enter your height (in metres): 1.8
You are 5.9055 feet tall.
```

Errors that happen while the program is running are called *exceptions*. The `try`/`except` block lets the program handle exceptions when they happen. If any exceptions happen while the `try` part is running, the `except` code is executed. It is ignored otherwise.

```
got_height = False

while not got_height:
    m_str = raw_input("Enter your height (in metres): ")
    try:
        metres = float(m_str)
        got_height = True # if we're here, it was converted.
    except:
        print "Please enter a number."

feet = 39.37 * metres / 12
print "You are " + str(feet) + " feet tall."
```

Figure 4.7: Asking until we get correct input

Figure 4.7 shows another example. In this program, the `while` loop will continue until there is no exception. The variable `got_height` is used to keep track of whether or not we have the input we need.

Check-Up Question

- Take a program you have written previously in this course that takes numeric input and modify it so it gives a nice error message.

Summary

This unit covers a lot of bits and pieces that don't necessarily let your programs *do* any more, but help you write programs that are better organized and are thus easier to maintain.

The modules in Python are very useful. In this course, we will try to point out relevant modules when you need them; we don't expect you to comb through the entire list of built-in modules every time you need something.

Key Terms

- functions
- arguments

- return value
- docstring
- variable scope
- module
- import
- object
- class
- property
- method
- exception