12. Modules

A **module** is a file containing Python definitions and statements intended for use in other Python programs. There are many Python modules that come with Python as part of the **standard library**. We have seen at least two of these already, the `turtle` module and the `string` module.

We have also shown you how to access help. The help system contains a listing of all the standard modules that are available with Python. Play with help!

12.1. Random numbers

We often want to use random numbers in programs, here are a few typical uses:

- To play a game of chance where the computer needs to throw some dice, pick a number, or flip a coin,
- To shuffle a deck of playing cards randomly,
- To allow/make an enemy spaceship appear at a random location and start shooting at the player,
- To simulate possible rainfall when we make a computerized model for estimating the environmental impact of building a dam,
- For encrypting banking sessions on the Internet.

Python provides a module `random` that helps with tasks like this. You can look it up using help, but here are the key things we'll do with it:

```
import random

# Create a black box object that generates random numbers
rng = random.Random()

dice_throw = rng.randrange(1,7)   # Return an int, one of 1,2,3,4,5,6
delay_in_seconds = rng.random() * 5.0
```

The `randrange` method call generates an integer between its lower and upper argument, using the same semantics as `range` — so the lower bound is included, but the upper bound is excluded. All the values have an equal probability of occurring (i.e. the results are *uniformly* distributed). Like `range`, `randrange` can also take an optional step argument. So let's assume we needed a random odd number less than 100, we could say:

```
r_odd = rng.randrange(1, 100, 2)
```

Other methods can also generate other distributions e.g. a bell-shaped, or "normal" distribution might be more appropriate for estimating seasonal rainfall, or the concentration of a compound in the body after taking a dose of medicine.

The `random` method returns a floating point number in the interval [0.0, 1.0) — the square bracket means "closed interval on the left" and the round parenthesis means "open interval on the right". In other words, 0.0 is possible, but all returned numbers will be strictly less than 1.0. It is usual to *scale* the results after calling this method, to get them into an interval suitable for your application. In the case shown here, we've converted the result of the method call to a number in the interval [0.0, 5.0). Once more, these are uniformly distributed numbers — numbers close to 0 are just as likely to occur as numbers close to 0.5, or numbers close to 1.0.

This example shows how to shuffle a list. (`shuffle` cannot work directly with a lazy promise, so notice that we had to convert the range object using the `list` type converter first.)

```
cards = list(range(52))  # Generate ints [0 .. 51]
                         #    representing a pack of cards.
rng.shuffle(cards)       # Shuffle the pack
```

12.1.1. Repeatability and Testing

Random number generators are based on a **deterministic** algorithm — repeatable and predictable. So they're called **pseudo-random** generators — they are not genuinely random. They start with a *seed* value. Each time you ask for another random number, you'll get one based on the current seed attribute, and the state of the seed (which is one of the attributes of the generator) will be updated.

For debugging and for writing unit tests, it is convenient to have repeatability — programs that do the same thing every time they are run. We can arrange this by forcing the random number generator to be initialized with a known seed every time. (Often this is only wanted during testing — playing a game of cards where the shuffled deck was always in the same order as last time you played would get boring very rapidly!)

```
drng = random.Random(123)  # Create generator with known starting state
```

This alternative way of creating a random number generator gives an explicit seed value to the object. Without this argument, the system probably uses something based on the time. So grabbing some random numbers from `drng` today will give you precisely the same random sequence as it will tomorrow!

12.1.2. Picking balls from bags, throwing dice, shuffling a pack of cards

Here is an example to generate a list containing *n* random ints between a lower and an upper bound:

```
import random

def make_random_ints(num, lower_bound, upper_bound):
    """
      Generate a list containing num random ints between lower_bound
      and upper_bound. upper_bound is an open bound.
    """
    rng = random.Random()  # Create a random number generator
    result = []
    for i in range(num):
        result.append(rng.randrange(lower_bound, upper_bound))
    return result
```

```
>>> make_random_ints(5, 1, 13)  # Pick 5 random month numbers
[8, 1, 8, 5, 6]
```

Notice that we got a duplicate in the result. Often this is wanted, e.g. if we throw a die five times, we would expect some duplicates.

But what if you don't want duplicates? If you wanted 5 distinct months, then this algorithm is wrong. In this case a good algorithm is to generate the list of possibilities, shuffle it, and slice off the number of elements you want:

```
xs = list(range(1,13))  # Make list 1..12  (there are no duplicates)
rng = random.Random()   # Make a random number generator
rng.shuffle(xs)         # Shuffle the list
result = xs[:5]         # Take the first five elements
```

In statistics courses, the first case — allowing duplicates — is usually described as pulling balls out of a bag *with replacement* — you put the drawn ball back in each time, so it can occur again. The latter case, with no duplicates, is usually described as pulling balls out of the bag *without replacement*. Once the ball is drawn, it doesn't go back to be drawn again. TV lotto games work like this.

The second "shuffle and slice" algorithm would not be so great if you only wanted a few elements, but from a very large domain. Suppose I wanted five numbers between one and ten million, without

duplicates. Generating a list of ten million items, shuffling it, and then slicing off the first five would be a performance disaster! So let us have another try:

```
import random

def make_random_ints_no_dups(num, lower_bound, upper_bound):
    """
      Generate a list containing num random ints between
      lower_bound and upper_bound. upper_bound is an open bound.
      The result list cannot contain duplicates.
    """
    result = []
    rng = random.Random()
    for i in range(num):
         while True:
             candidate = rng.randrange(lower_bound, upper_bound)
             if candidate not in result:
                 break
         result.append(candidate)
    return result

xs = make_random_ints_no_dups(5, 1, 10000000)
print(xs)
```

This agreeably produces 5 random numbers, without duplicates:

```
[3344629, 1735163, 9433892, 1081511, 4923270]
```

Even this function has its pitfalls. Can you spot what is going to happen in this case?

```
xs = make_random_ints_no_dups(10, 1, 6)
```

12.2. The `time` module

As we start to work with more sophisticated algorithms and bigger programs, a natural concern is *"is our code efficient?"* One way to experiment is to time how long various operations take. The `time` module has a function called `clock` that is recommended for this purpose. Whenever `clock` is called, it returns a floating point number representing how many seconds have elapsed since your program started running.

The way to use it is to call `clock` and assign the result to a variable, say `t0`, just before you start executing the code you want to measure. Then after execution, call `clock` again, (this time we'll save the result in variable `t1`). The difference `t1-t0` is the time elapsed, and is a measure of how fast your program is running.

Let's try a small example. Python has a built-in `sum` function that can sum the elements in a list. We can also write our own. How do we think they would compare for speed? We'll try to do the summation of a list [0, 1, 2 ...] in both cases, and compare the results:

```
import time

def do_my_sum(xs):
    sum = 0
    for v in xs:
        sum += v
    return sum

sz = 10000000        # Lets have 10 million elements in the list
testdata = range(sz)

t0 = time.clock()
my_result = do_my_sum(testdata)
t1 = time.clock()
print("my_result    = {0} (time taken = {1:.4f} seconds)"
        .format(my_result, t1-t0))

```

```
t2 = time.clock()
their_result = sum(testdata)
t3 = time.clock()
print("their_result = {0} (time taken = {1:.4f} seconds)"
        .format(their_result, t3-t2))
```

On a reasonably modest laptop, we get these results:

```
my_sum    = 49999995000000 (time taken = 1.5567 seconds)
their_sum = 49999995000000 (time taken = 0.9897 seconds)
```

So our function runs about 57% slower than the built-in one. Generating and summing up ten million elements in under a second is not too shabby!

12.3. The `math` module

The `math` module contains the kinds of mathematical functions you'd typically find on your calculator (`sin`, `cos`, `sqrt`, `asin`, `log`, `log10`) and some mathematical constants like `pi` and `e`:

```
>>> import math
>>> math.pi                 # Constant pi
3.141592653589793
>>> math.e                  # Constant natural log base
2.718281828459045
>>> math.sqrt(2.0)          # Square root function
1.4142135623730951
>>> math.radians(90)        # Convert 90 degrees to radians
1.5707963267948966
>>> math.sin(math.radians(90))  # Find sin of 90 degrees
1.0
>>> math.asin(1.0) * 2      # Double the arcsin of 1.0 to get pi
3.141592653589793
```

Like almost all other programming languages, angles are expressed in *radians* rather than degrees. There are two functions `radians` and `degrees` to convert between these two popular ways of measuring angles.

Notice another difference between this module and our use of `random` and `turtle`: in `random` and `turtle` we create objects and we call methods on the object. This is because objects have *state* — a turtle has a color, a position, a heading, etc., and every random number generator has a seed value that determines its next result.

Mathematical functions are "pure" and don't have any state — calculating the square root of 2.0 doesn't depend on any kind of state or history about what happened in the past. So the functions are not methods of an object — they are simply functions that are grouped together in a module called `math`.

12.4. Creating your own modules

All we need to do to create our own modules is to save our script as a file with a `.py` extension. Suppose, for example, this script is saved as a file named `seqtools.py`:

```
def remove_at(pos, seq):
    return seq[:pos] + seq[pos+1:]
```

We can now use our module, both in scripts we write, or in the interactive Python interpreter. To do so, we must first `import` the module.

```
>>> import seqtools
>>> s = "A string!"
>>> seqtools.remove_at(4, s)
'A sting!'
```

We do not include the `.py` file extension when importing. Python expects the file names of Python modules to end in `.py`, so the file extension is not included in the **import statement**.

The use of modules makes it possible to break up very large programs into manageable sized parts, and to keep related parts together.

12.5. Namespaces

A **namespace** is a collection of identifiers that belong to a module, or to a function, (and as we will see soon, in classes too). Generally, we like a namespace to hold "related" things, e.g. all the math functions, or all the typical things we'd do with random numbers.

Each module has its own namespace, so we can use the same identifier name in multiple modules without causing an identification problem.

```
# Module1.py

question = "What is the meaning of Life, the Universe, and Everything?"
answer = 42
```

```
# Module2.py

question = "What is your quest?"
answer = "To seek the holy grail."
```

We can now import both modules and access `question` and `answer` in each:

```
import module1
import module2

print(module1.question)
print(module2.question)
print(module1.answer)
print(module2.answer)
```

will output the following:

```
What is the meaning of Life, the Universe, and Everything?
What is your quest?
42
To seek the holy grail.
```

Functions also have their own namespaces:

```
def f():
    n = 7
    print("printing n inside of f:", n)

def g():
    n = 42
    print("printing n inside of g:", n)

n = 11
print("printing n before calling f:", n)
f()
print("printing n after calling f:", n)
g()
print("printing n after calling g:", n)
```

Running this program produces the following output:

```
printing n before calling f: 11
printing n inside of f: 7
printing n after calling f: 11
printing n inside of g: 42
printing n after calling g: 11
```

The three `n`'s here do not collide since they are each in a different namespace — they are three names for three different variables, just like there might be three different instances of people, all called "Bruce".

Namespaces permit several programmers to work on the same project without having naming collisions.

How are namespaces, files and modules related?

Python has a convenient and simplifying one-to-one mapping, one module per file, giving rise to one namespace. Also, Python takes the module name from the file name, and this becomes the name of the namespace. `math.py` is a filename, the module is called `math`, and its namespace is `math`. So in Python the concepts are more or less interchangeable.

But you will encounter other languages (e.g. C#), that allow one module to span multiple files, or one file to have multiple namespaces, or many files to all share the same namespace. So the name of the file doesn't need to be the same as the namespace.

So a good idea is to try to keep the concepts distinct in your mind.

Files and directories organize *where* things are stored in our computer. On the other hand, namespaces and modules are a programming concept: they help us organize how we want to group related functions and attributes. They are not about "where" to store things, and should not have to coincide with the file and directory structures.

So in Python, if you rename the file `math.py`, its module name also changes, your `import` statements would need to change, and your code that refers to functions or attributes inside that namespace would also need to change.

In other languages this is not necessarily the case. So don't blur the concepts, just because Python blurs them!

12.6. Scope and lookup rules

The **scope** of an identifier is the region of program code in which the identifier can be accessed, or used.

There are three important scopes in Python:

- **Local scope** refers to identifiers declared within a function. These identifiers are kept in the namespace that belongs to the function, and each function has its own namespace.
- **Global scope** refers to all the identifiers declared within the current module, or file.
- **Built-in scope** refers to all the identifiers built into Python — those like `range` and `min` that can be used without having to import anything, and are (almost) always available.

Python (like most other computer languages) uses precedence rules: the same name could occur in more than one of these scopes, but the innermost, or local scope, will always take precedence over the global scope, and the global scope always gets used in preference to the built-in scope. Let's start with a simple example:

```
1  def range(n):
2      return 123*n
3
4  print(range(10))
```

What gets printed? We've defined our own function called `range`, so there is now a potential ambiguity. When we use `range`, do we mean our own one, or the built-in one? Using the scope lookup rules determines this: our own `range` function, not the built-in one, is called, because our function `range` is in the global namespace, which takes precedence over the built-in names.

So although names likes `range` and `min` are built-in, they can be "hidden" from your use if you choose to define your own variables or functions that reuse those names. (It is a confusing practice to redefine built-in names — so to be a good programmer you need to understand the scope rules and understand that you can do nasty things that will cause confusion, and then you avoid doing them!)

Now, a slightly more complex example:

```
1 n = 10
2 m = 3
3 def f(n):
4     m = 7
5     return 2*n+m
6
7 print(f(5), n, m)
```

This prints 17 10 3. The reason is that the two variables `m` and `n` in lines 1 and 2 are outside the function in the global namespace. Inside the function, new variables called `n` and `m` are created *just for the duration of the execution of f*. These are created in the local namespace of function `f`. Within the body of `f`, the scope lookup rules determine that we use the local variables `m` and `n`. By contrast, after we've returned from `f`, the `n` and `m` arguments to the `print` function refer to the original variables on lines 1 and 2, and these have not been changed in any way by executing function `f`.

Notice too that the `def` puts name `f` into the global namespace here. So it can be called on line 7.

What is the scope of the variable `n` on line 1? Its scope — the region in which it is visible — is lines 1, 2, 6, 7. It is hidden from view in lines 3, 4, 5 because of the local variable `n`.

12.7. Attributes and the dot operator

Variables defined inside a module are called **attributes** of the module. We've seen that objects have attributes too: for example, most objects have a `__doc__` attribute, some functions have a `__annotations__` attribute. Attributes are accessed using the **dot operator** (`.`). The `question` attribute of `module1` and `module2` is accessed using `module1.question` and `module2.question`.

Modules contain functions as well as attributes, and the dot operator is used to access them in the same way. `seqtools.remove_at` refers to the `remove_at` function in the `seqtools` module.

When we use a dotted name, we often refer to it as a **fully qualified name**, because we're saying exactly which `question` attribute we mean.

12.8. Three `import` statement variants

Here are three different ways to import names into the current namespace, and to use them:

```
1 import math
2 x = math.sqrt(10)
```

Here just the single identifier `math` is added to the current namespace. If you want to access one of the functions in the module, you need to use the dot notation to get to it.

Here is a different arrangement:

```
1 from math import cos, sin, sqrt
2 x = sqrt(10)
```

The names are added directly to the current namespace, and can be used without qualification. The name `math` is not itself imported, so trying to use the qualified form `math.sqrt` would give an error.

Then we have a convenient shorthand:

```
1 from math import *   # Import all the identifiers from math,
2                      #   adding them to the current namespace.
3 x = sqrt(10)         # Use them without qualification.
```

Of these three, the first method is generally preferred, even though it means a little more typing each time. Although, we can make things shorter by importing a module under a different name:

```
>>> import math as m
>>> m.pi
3.141592653589793
```

But hey, with nice editors that do auto-completion, and fast fingers, that's a small price!

Finally, observe this case:

```
def area(radius):
    import math
    return math.pi * radius * radius

x = math.sqrt(10)      # This gives an error
```

Here we imported `math`, but we imported it into the local namespace of `area`. So the name is usable within the function body, but not in the enclosing script, because it is not in the global namespace.

12.9. Turn your unit tester into a module

Near the end of Chapter 6 (Fruitful functions) we introduced unit testing, and our own `test` function, and you've had to copy this into each module for which you wrote tests. Now we can put that definition into a module of its own, say `unit_tester.py`, and simply use one line in each new script instead:

```
from unit_tester import test
```

12.10. Glossary

attribute

A variable defined inside a module (or class or instance – as we will see later). Module attributes are accessed by using the **dot operator** (`.`).

dot operator

The dot operator (`.`) permits access to attributes and functions of a module (or attributes and methods of a class or instance – as we have seen elsewhere).

fully qualified name

A name that is prefixed by some namespace identifier and the dot operator, or by an instance object, e.g. `math.sqrt` or `tess.forward(10)`.

import statement

A statement which makes the objects contained in a module available for use within another module. There are two forms for the import statement. Using hypothetical modules named `mymod1` and `mymod2` each containing functions `f1` and `f2`, and variables `v1` and `v2`, examples of these two forms include:

```
import mymod1
from mymod2 import f1, f2, v1, v2
```

The second form brings the imported objects into the namespace of the importing module, while the first form preserves a separate namespace for the imported module, requiring `mymod1.v1` to access the `v1` variable from that module.

method

Function-like attribute of an object. Methods are *invoked* (called) on an object using the dot operator. For example:

```
>>> s = "this is a string."
>>> s.upper()
```

```
'THIS IS A STRING.'
>>>
```

We say that the method, `upper` is invoked on the string, `s`. `s` is implicitely the first argument to `upper`.

module

A file containing Python definitions and statements intended for use in other Python programs. The contents of a module are made available to the other program by using the `import` statement.

namespace

A syntactic container providing a context for names so that the same name can reside in different namespaces without ambiguity. In Python, modules, classes, functions and methods all form namespaces.

naming collision

A situation in which two or more names in a given namespace cannot be unambiguously resolved. Using

```
import string
```

instead of

```
from string import *
```

prevents naming collisions.

ndard library A library is a collection of software used as tools in the development of other software. The standard library of a programming language is the set of such tools that are distributed with the core programming language. Python comes with an extensive standard library.

12.11. Exercises

1. Open help for the `calendar` module.

    a. Try the following:

    ```
    import calendar
    cal = calendar.TextCalendar()      # Create an instance
    cal.pryear(2012)                   # What happens here?
    ```

    b. Observe that the week starts on Monday. An adventurous CompSci student believes that it is better mental chunking to have his week start on Thursday, because then there are only two working days to the weekend, and every week has a break in the middle. Read the documentation for TextCalendar, and see how you can help him print a calendar that suits his needs.

    c. Find a function to print just the month in which your birthday occurs this year.

    d. Try this:

    ```
    d = calendar.LocaleTextCalendar(6, "SPANISH")
    d.pryear(2012)
    ```

    Try a few other languages, including one that doesn't work, and see what happens.

    e. Experiment with `calendar.isleap`. What does it expect as an argument? What does it return as a result? What kind of a function is this?

Make detailed notes about what you learned from these exercises.

2. Open help for the `math` module.
   a. How many functions are in the `math` module?
   b. What does `math.ceil` do? What about `math.floor`? (*hint:* both `floor` and `ceil` expect floating point arguments.)
   c. Describe how we have been computing the same value as `math.sqrt` without using the `math` module.
   d. What are the two data constants in the `math` module?

   Record detailed notes of your investigation in this exercise.
3. Investigate the `copy` module. What does `deepcopy` do? In which exercises from last chapter would `deepcopy` have come in handy?
4. Create a module named `mymodule1.py`. Add attributes `myage` set to your current age, and `year` set to the current year. Create another module named `mymodule2.py`. Add attributes `myage` set to 0, and `year` set to the year you were born. Now create a file named `namespace_test.py`. Import both of the modules above and write the following statement:

```
print( (mymodule2.myage - mymodule1.myage) ==
       (mymodule2.year - mymodule1.year)  )
```

   When you will run `namespace_test.py` you will see either `True` or `False` as output depending on whether or not you've already had your birthday this year.

   What this example illustrates is that out different modules can both have attributes named `myage` and `year`. Because they're in different namespaces, they don't clash with one another. When we write `namespace_test.py`, we fully qualify exactly which variable `year` or `myage` we are referring to.
5. Add the following statement to `mymodule1.py`, `mymodule2.py`, and `namespace_test.py` from the previous exercise:

```
print("My name is", __name__)
```

   Run `namespace_test.py`. What happens? Why? Now add the following to the bottom of `mymodule1.py`:

```
if __name__ == "__main__":
    print("This won't run if I'm  imported.")
```

   Run `mymodule1.py` and `namespace_test.py` again. In which case do you see the new print statement?
6. In a Python shell / interactive interpreter, try the following:

```
>>> import this
```

   What does Tim Peters have to say about namespaces?
7. Give the Python interpreter's response to each of the following from a continuous interpreter session:

```
>>> s = "If we took the bones out, it wouldn't be crunchy, would it?"
>>> s.split()
>>> type(s.split())
>>> s.split("o")
>>> s.split("i")
>>> "0".join(s.split("o"))
```

   Be sure you understand why you get each result. Then apply what you have learned to fill in the body of the function below using the `split` and `join` methods of `str` objects:

```
def myreplace(old, new, s):
    """ Replace all occurrences of old with new in s. """
    ...


test(myreplace(",", ";", "this, that, and some other thing") ==
                         "this; that; and some other thing")
test(myreplace(" ", "**",
               "Words will now      be  separated by stars.") ==
               "Words**will**now**be**separated**by**stars.")
```

Your solution should pass the tests.

8. Create a module named `wordtools.py` with our test scaffolding in place.

   Now add functions to these tests pass:

```
test(cleanword("what?") == "what")
test(cleanword("'now!'") == "now")
test(cleanword("?+='w-o-r-d!,@$()'") ==  "word")

test(has_dashdash("distance--but"))
test(not has_dashdash("several"))
test(has_dashdash("spoke--"))
test(has_dashdash("distance--but"))
test(not has_dashdash("-yo-yo-"))

test(extract_words("Now is the time!  'Now', is the time? Yes, now.") ==
      ['now','is','the','time','now','is','the','time','yes','now'])
test(extract_words("she tried to curtsey as she spoke--fancy") ==
      ['she','tried','to','curtsey','as','she','spoke','fancy'])

test(wordcount("now", ["now","is","time","is","now","is","is"]) == 2)
test(wordcount("is", ["now","is","time","is","now","the","is"]) == 3)
test(wordcount("time", ["now","is","time","is","now","is","is"]) == 1)
test(wordcount("frog", ["now","is","time","is","now","is","is"]) == 0)

test(wordset(["now", "is", "time", "is", "now", "is", "is"]) ==
      ["is", "now", "time"])
test(wordset(["I", "a", "a", "is", "a", "is", "I", "am"]) ==
      ["I", "a", "am", "is"])
test(wordset(["or", "a", "am", "is", "are", "be", "but", "am"]) ==
      ["a", "am", "are", "be", "but", "is", "or"])

test(longestword(["a", "apple", "pear", "grape"]) == 5)
test(longestword(["a", "am", "I", "be"]) == 2)
test(longestword(["this","supercalifragilisticexpialidocious"]) == 34)
test(longestword([ ]) == 0)
```

Save this module so you can use the tools it contains in future programs.