

Functionally Modular

Self-Review Questions

Self-review 5.1 Which names are *local*, which are *global* and which are *built-in* in the following code fragment?

```
space_invaders = [ ... ]
player_pos = ( 200 , 25 )
level = 1
max_level = 10

def play ( ) :
    ...
    while ( level < max_level + 1 ) :
        if len ( space_invaders ) == 0 :
            level += 1
            continue
    ...
```

Global names:

- space_invaders
- player_pos
- level
- max_level

Built-in names:

- len

... and there are no local names.

Self-review 5.2 What is special about a *recursive* function?

Self-review 5.3 What is a *side effect*?

Self-review 5.4 What does the term *referential transparency* mean?

A function is *referentially transparent* if it returns the same result every time you execute it with the same arguments. In programming languages such as Python, this is the same as saying that the function does not update any variables – or that the function has no *side effects*. So, for example the following function is referentially transparent:

```
def sum (l):
    if l == []: return 0
    else: return 1 + sum (l[1:])
```

but this next function is not referentially transparent, because it updates the variable a:

```
a = 10
def example ():
    global a
    a = a + 1
    return a * 10
```

Self-review 5.5 What list of lists does the following list comprehension evaluate to?

```
[['#' for col in range (3)] for row in range (3)]
```

Self-review 5.6 What is the result of executing the following code?

```
def dp (l1, l2):
    def p (l1, l2, n):
        return l1[n] * l2[n]
    r = 0
    for i in range (len (l1)):
        r += p (l1, l2, i)
    return r

print dp ([1, 2, 3], [4, 5, 6])
```

This code prints 32.

The function `p` multiplies two elements in a particular index in two given lists. So, for example `p ([1, 2, 3], [4, 5, 6], 1)` will return the product of the second elements in the two lists, which is $2 * 5$. The outer function `dp` uses `p` to multiply all the corresponding elements of the two lists in the arguments and totals the results. So, for the two lists in the function call, we get $(1 * 4) + (2 * 5) + (3 * 6)$ which is $4 + 10 + 18$, or 32.

The function `dp` is named after the idea of a “dot product” in mathematics – this is a very useful concept that is used in applications such as games and image processing.

Self-review 5.7 In Python modules, what is the following code idiom used for?

```
if __name__ == '__main__':
    ...
```

Self-review 5.8 What is wrong with the following code?

```
from turtle import *
turtle.forward (100)
```

Self-review 5.9 What are the following special variables used for?

1. `__author__`

2. `__date__`
3. `__copyright__`
4. `__version__`
5. `__revision__`
6. `__license__`
7. `__credits__`

Self-review 5.10 What is the result of executing the following Python code?

```
l = [ i for i in range ( 1 , 100 ) ]
map ( lambda x : x ** 3 , l )
```

Is there a better way of expressing the algorithm in Python?

The list `l` contains the integers from 1 to 99. The `map` function returns a list of integers which are the cubes of the list `l`, so `[1 * 1 * 1 , 2 * 2 * 2 , 3 * 3 * 3 , ...]` or `[1 , 8 , 27 , 64 , 125 , ...]`.

There are several ways to do this in Python, but one way is to use list comprehensions:

```
cubes = [ i ** 3 for i in range ( 1 , 100 ) ]
```

Self-review 5.11 What is the result of the following Python code?

```
import string
v = [ 'a' , 'e' , 'i' , 'o' , 'u' , 'A' , 'E' , 'I' , 'O' , 'U' ]
filter ( lambda x : x in string.uppercase , v )
```

Self-review 5.12 What is a generator in Python?

Self-review 5.13 What does the Python keyword `yield` mean?

Self-review 5.14 What does the following Python function do?

```
def geni ( n ) :
    for i in range ( n ) :
        yield i
```

The function `geni` returns a *generator* which generates the integers from 1 to `n`. For example:

```
>>> def geni ( n ) :
...     for i in range ( n ) :
...         yield i
...
>>> g = geni ( 10 )
>>> g.next ( )
0
>>> g.next ( )
1
>>> g.next ( )
2
>>> g.next ( )
```

```

3
>>> g.next ()
4
>>> g.next ()
5
>>>

```

Self-review 5.15 Using the documentation on the Python website, find out what the exception `StopIteration` is used for.

Programming Exercises

Exercise 5.1 The following Python code represents a Tic-Tac-Toe board as a list of lists:

```
[[ '#', 'o', 'x' ], [ '#', '#', 'o' ], [ 'x', '#', 'o' ]]
```

The `#` symbols represent blank squares on the board. Write a function `print_board` that takes a list of lists as an argument and prints out a Tic-Tac-Toe board in the following format:

```

  | o | x
-----
  | | o
-----
x | | o

```

```

def print_board ( board ):
    for row in ( 0, 1, 2 ):
        for col in ( 0, 1, 2 ):
            if board[row][col] == '#':
                print ' ',
            else : print board[row][col],
                if col < 2 : print ' | ',
            if row < 2 : print '\n-----'
    print # Print an empty line

```

Exercise 5.2 Convert the following iterative functions into recursive functions:

1.

```

def sum_even ( n ):
    total = 0
    for i in range ( 2, n + 1, 2 ):
        total += i
    return total

```
2.

```

def min ( l ):
    m = 0
    for i in l :
        if i < m : m = i
    return m

```

Any iterative function can be recoded as a recursive function, and any recursive function can be recoded as an iterative function.

```

3.     def prod ( l ) :
        product , i = 1 , 0
        while i < len ( l ) :
            product *= l[i]
            i += 1
        return product

```

Hint: You may find it useful to add an extra parameter to the recursive version of min.

Exercise 5.3 Convert the following recursive functions into iterative ones:

```

1.     def sum_odd ( n , total ) :
        if n == 1 : return total
        elif n % 2 == 0 : return sum_odd ( n - 1 , total )
        else : return sum_odd ( n - 2 , total + n )

2.     def max ( l , n ) :
        if l == [] : return n
        elif l[0] > n : return max ( l[1:] , l[0] )
        else : return max ( l[1:] , n )

3.     def mylen ( l , n ) :
        if l == [] : return n
        else : return mylen ( l[1:] , n + 1 )

```

```

def sum_odd ( n ) :
    total = 0
    for i in range ( n + 1 ) :
        if i % 2 != 0 :
            total += i
    return total

```

```

def max ( l ) :
    m = 0
    if l == [] : return None # Dummy value
    else : m = l[0]
    for i in l :
        if i > m : m = i
    return m

```

```

def mylen ( l ) :
    if l == [] : return 0
    else : total = 0
    for i in l :
        total += 1
    return total

```

Exercise 5.4 The following code is a module that provides functions for drawing some simple shapes using the Python Turtle module. Copy the code into a file called `shape.py` and add in:

- Documentation for the module and its functions.
- Special variables such as `__date__`.

- Some simple testing.

Make sure you use pydoc to generate a webpage containing your documentation.

```
__author__ = 'Sarah Mount'

from turtle import *

def square ( n ) :
    for i in range ( 4 ) :
        forward ( n )
        left ( 90 )

def rectangle ( s1 , s2 ) :
    for i in range ( 2 ) :
        forward ( s1 )
        left ( 90 )
        forward ( s2 )
        left ( 90 )

def pentagon ( s ) :
    for i in range ( 5 ) :
        forward ( s )
        left ( 360 / 5 )
```

Exercise 5.5 Use map and **lambda** to turn a list of integers from 1 to 100 into a list of even numbers from 2 to 200.

```
map ( lambda x : x * 2 , range ( 1 , 101 ) )
```

Exercise 5.6 Use filter to generate a list of odd numbers from 0 to 100.

Exercise 5.7 Use a list comprehension to generate a list of odd numbers from 0 to 100.

```
[ i for i in range ( 101 ) if i % 2 != 0 ]
```

Exercise 5.8 Write a generator function (using the yield keyword) that generates factorial numbers.

Exercise 5.9 Ackermann's Function is defined as:

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Write a recursive Python function to implement Ackermann's Function. How many recursive calls will be required to evaluate $A(2, 3)$?

Exercise 5.10 In Section ?? (page ??) we introduced palindromes as words or phrases that read the same forwards and backwards, and showed implementations using iteration and negative indexing. For this question we say that 'deed', 'peep' and 'a man, a plan, a canal, panama!' are all palindromes – so we do not make spaces significant. Write a recursive function implementation of `isPalindrome` to test whether or not a string is a palindrome.

Exercise 5.11 The first five rows of Pascal's Triangle are:

```

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 . . .

```

Each number is generated by adding the two above it. Write a recursive function to generate the first n lines of Pascal's Triangle.

Challenges

Challenge 5.1 Create a module for playing Tic-Tac-Toe.

Hint: You may want to consider the following functions:

1. `print_board ()` – from the programming exercise above, except that you will want to use a global `board` variable, rather than a function argument.
2. `has_won ()` – check to see whether either player has won. This function should return the string 'o' or 'x' if either the o or x player has won and '#' if neither player has won. A player wins in Tic-Tac-Toe if they have three of their counters in a row, a column or a diagonal.
3. `place_counter (sq, counter)` – place the `counter` on a particular square of the board. The first argument should be a number between 0 and 8 and the second argument should be either 'o' or 'x'. You should consider the squares on the board to be numbered as in the diagram below.

```

  0 | 1 | 2
  -----
  3 | 4 | 5
  -----
  6 | 7 | 8

```

Using a numbering such as this one makes the answer to this challenge simpler!

4. `next_play ()` – This function should ask the user for the next move they want to play. You should make sure that the user knows whether x or o is currently playing. You can assume that the user will enter an integer value. You should still check that the integer the player has provided is between 0 and 8 inclusive.

5. `play()` – This is the top-level function that tells Python how to play Tic-Tac-Toe. The algorithm for game play is:

- If one of the players has won the game, print a message on the screen to congratulate them.
- If no-one has won the game, then:
 - Use the `next_play` function to get the current player's next move.
 - Play the next move by calling the `place_counter` function.
 - Print the new board.
 - Change the current player – either from 'x' to 'o' or from 'o' to 'x'.

You will also need two global variables, one to store the current state of the board and one to say which player is playing next.

Below is an outline of the module to get you started – make sure you add some documentation and make good use of Python's special variables, like `__author__` and `__version__`.

```
board = [['#' for col in range(3)] for row in range(3)]
o_playing = True
def play():
    # Your code here!
def next_play():
    # Your code here!
def has_won():
    # Your code here!
def place_counter():
    # Your code here!
def print_board():
    # Your code here!

if __name__ == '__main__':
    play()
```

Challenge 5.2 One reason for selecting a recursive version of an algorithm over an iterative one, or vice versa, is running time. There are various ways of assessing whether one algorithm is faster than another. Big O notation is a very important one – we will come to this in Chapter ???. A technique for assessing the speed of programs implementing algorithms is to executing benchmarks. Benchmarks require us to be able to time the execution of a program. Python helps us here, as it provides a module called `timeit` that can be used to time the execution of Python expressions. Here is an example of the `timeit` module in use:

```
>>> import timeit
>>>
>>> def is_prime(n):
...     return not bool(filter(lambda x: n%x == 0, range(2, n)))
...
>>> def primes(n):
...     return filter(is_prime, range(2, n))
```

```
...
>>> t = timeit.Timer ('primes ( 100)', 'from __main__ import primes')
>>> print t.timeit ( 500 )
1.47258400917
>>>
```

Here, we have written some code to time one of our prime number generators, and found out how long it has taken Python to execute the code 500 times. We did this by creating an object called `timeit.Timer` that took a call to our `primes` function as its first argument. The second argument, `'from __main__ import primes'` tells the `timeit` module where it can find the `primes` function – in this case in the `__main__` namespace.

The argument to the `timeit` method, `500`, means 'execute the code 500 times'. Python code generally executes very quickly, so timing a very short piece of code that only runs once can be inaccurate. So, instead, we can ask `timeit` to time the code over several hundred or thousand runs, which will give us more accurate information. In the example above, the code took `1.47258400917 s` to run.

For this challenge, test the time of execution of all the prime number generators we have presented in this book, timing them using the `timeit` module, to determine which is the fastest.

Challenge 5.3 Chess is played on an 8×8 board. A *queen* may attack any piece on the same row, column or diagonal as herself.

The Eight Queens Problem involves placing eight queens on a chess board in such a way that no queen is attacking any other – so, there must be at most one queen on any row, column or diagonal.

There are 92 solutions to the Eight Queens Problem and the challenge is to write a program that finds all of them.

Hint: Think about how to represent the board. A list of lists of Booleans might be a sensible choice, but are there other options? Whatever you use, you will probably want to make sure that you can print chess boards in a human-readable format, such as this:

```
Q x x x x x x
x x Q x x x x
x x x x Q x x
x Q x x x x x
x x x x x x Q
x x x Q x x x
x x x x x Q x
x x x x Q x x
```

The Eight Queens Problem is the subject of the classic paper, *Program Development by Stepwise Refinement* by Nicklaus Wirth. You can find his paper at the ACM 'classics' website: <http://www.acm.org/classics/dec95/>

Index

Ackermann's Function, 40

Eight Queens Problem, 44

palindrome, 42

Pascal's Triangle, 42