> All right … all right … but apart from better sanitation and
>
> medicine and education and irrigation and public health and roads and a
>
> freshwater system and baths and public order … what have the Romans
>
> ever done for us?
>
> — Monty Python's Life of Brian

Functional Decomposition

Reusing the Code

In the previous readings we used control structures to implement a simple conversation between a user and the computer. It is not very much to ask the user for all the topics to discuss. Let's add in some code that starts off with a query about university, and make the computer's reply "That's very interesting". In principle, this is easy, because we already have code that does a discussion, so perhaps we can reuse it by just copying it and changing certain parts.

```
name = input("What is your name? ")
if name == "Tim" :
    print("Greetings, Tim the Enchanter")
elif name == "Brian" :
    print("Bad luck, Brian")
else :
    print("Hello " + name + ".")

like = input("Do you like university? ")
response = input("Why do you think that? ")
print("That's very interesting.")

while True :
    topic = input("What do you want to talk about? ")
    if topic == "nothing" :
        break
    like = input("Do you like " + topic + "? ")
    response = input("Why do you think that? ")
    print("I also think that", response)

print("Okay. Goodbye, " + name + "!")
```

This works, but there are a couple of things that might go wrong with copying code like this. First, it makes the code look more complicated than it is, and it is harder to read when the code is in two places instead of one. The second issue is to do with updating the program: if we want to modify how a discussion works, then we would need to update the code in two separate places. Worse, if we forget one of those two places, then the two discussions become different.

What we want to be able to do is to pull out, or abstract the idea of a "discussion" into a separate part of the code, and then be able to easily use that code in as many places as possible. Ideally, we want to end up with code that behaves like this (with code instead of the comments):

```
# have a discussion on "university"
print("That's very interesting.")
while True :
    topic = input("What do you want to talk about? ")
    if topic == "nothing" :
        break
    # have a discussion on "topic"
    print("I also think that")# , user's response
```

And then somewhere else in the code, we can define how a discussion works. This abstraction is called a **function**. Here is what the definition of a “discuss” function looks like in Python:

```
def discuss(topic) :
    like = input("Do you like " + topic + "? ")
    response = input("Why do you think that? ")
    return response
```

We use the `def` keyword to say that we are defining the function called `discuss`. Inside parentheses, we state the *formal parameters*, also called *arguments*, that the function takes, which are the bits of information that the function needs to know to complete its task. In this case, we cannot discuss something without knowing what topic to discuss, so we specify that the `discuss` function takes a single parameter, which we will give the name `topic`. After the first line is an indented body of code, most of which we are already familiar with. In this case, the discussion needs to supply or ‘give back’ a result at the end, which is the response that the user entered. We do this using the **return** keyword, which is shown in the last line above. When a `return` statement is reached, the function will end immediately; if there is a value after the return keyword, it is the result that is ‘given back’ (we say the value is *returned*). Every time that we use the `discuss` function, this body of code is what actually gets executed. The indentation behaves in the same way we’ve seen before – when we stop indenting, then we have finished the function definition. Now that we have this abstraction, we can use the `discuss` function instead of using the actual discussion code directly:

```
name = input("What is your name? ")
if name == "Tim" :
    print("Greetings, Tim the Enchanter")
elif name == "Brian" :
    print("Bad luck, Brian")
else :
    print("Hello " + name + ".")

discuss("university")
print("That's very interesting.")

while True :
    topic = input("What do you want to talk about? ")
    if topic == "nothing" :
        break
    response = discuss(topic)
    print("I also think that", response)

print("Okay. Goodbye, " + name + "!")
```

The full program that we have written is in `interaction.py`. This file also includes comments, which are discussed in the next section.

Notice the use of the return value in this line: `response = discuss(topic)`. This will execute the `discuss` function above, and the value that is returned is assigned to the variable `response`. The line `discuss("university")` shows a situation where our function is used for its side-effect of interacting with the user. The value is still returned by the function, but it is discarded by the main program.

Function Syntax

A definition has the following form:

```
def function_name(arg1, arg2, ...) :
    body
```

Function definitions start with the word `def`, followed by a `function_name` which follows the same syntax rules as for variable names. In parentheses is a comma separated list of arguments – these are all names of variables and are usually called the **formal parameters**. The body is an indented sequence of statements.

Executing, or "calling", a function uses the syntax

```
function_name(value1, value2, ...)
```

The comma-separated values are called the **actual parameters**. The number of `values` must be the same as the number of `args` in the definition.

Semantics

When a function is called, the actual parameters are associated with the formal parameters. That is, each `arg` is treated as a variable with the corresponding `value`. Then the body is executed. After that, the program will continue from the point where the function was called. If a `return` statement is executed, then the function will exit immediately. The resulting value of the function call will be the value used in the `return` statement.

Another example of function definition and use Although addition is built into Python we can create our own definition as follows.

```
def add(n, m) :
    return n + m
```

In detail – `def` introduces the function definition for `add`. The name of the function is `add` and its formal parameters are `n` and `m`. The function simply

computes the sum of `n` and `m` and returns that as the result of the function (using the `return` statement).

Here are some example uses.

```
>>> add(2, 3)
5
>>> 3 * add(2, 3)
15
```

In the first case we call the function passing in actual parameters 2 and 3. When the function is evaluated, `n` gets the value 2 and `m` gets the value 3. The function returns 5 as is seen when evaluated in the interpreter. The second example shows that we can use a call to the function within an arithmetic expression.

Decomposing Problems

There is more to software engineering than knowing how to write code. Part of the design process discussed above is problem decomposition. If we are given a description of a problem or task, how can we write a program that solves the problem?

The problem we will work on in this section is “write a program to find the nth prime number”. The first prime is 2, the second is 3, the third is 5 and so on.

The first step (after we understand the problem) is to decompose the problem into simpler and simpler subproblems until we reach a point where the subproblems are easy to implement.

Given we need to find the nth prime number, we could start at some prime (like 2 or 3) and find the next prime number. If we repeat this process, we will get to the prime we want. So an interesting subproblem is “given a number n, find the next prime number after n”.

To do this we can repeatedly look at the next number and ask if that number is a prime, if so we have finished, if not we keep looking. So the next interesting subproblem is “given a number n determine if n is a prime”.

Recall that a prime is a number that is divisible by only itself and one. We can therefore test for a number being a prime if we can test for divisibility of every number from 2 up to n. So, the last subproblem is “given numbers n and m, determine if m divides n (exactly)”.

Divisible or not Divisible

There is a useful mathematical operation that can be used for testing for divisibility. The operator `%` is known as **mod** or **modulo**. It returns the remainder of division, making it useful to test for divisibility.

```
>>> 7 % 2
1
>>> 7 % 4
3
>>> 9 % 3
```

```
0
>>> 10 % 5 == 0
True
>>> 7 % 4 == 0
False
```

The first three results are because `7 // 2` is `3` with remainder `1`, `7 // 4` is `1` with remainder `3`, and `9 // 3` is `3` with remainder `0`. Testing for divisibility is the same as testing if the remainder is equal to `0`. So, `10` is divisible by `5`, and `7` is not divisible by `4`.

Now we have reached a level of detail that we know how to write it all in Python code, so we can now start writing the code, building up to larger subproblems.

Is a Number Prime?

We will write a function called `is_prime` to test a number and return whether it is a prime number or not, using the idea above of testing numbers from 2 to n. This code can be downloaded as the file `is_prime1.py`.

```
def is_prime(num) :
    """Returns True iff 'num' is prime.

    Parameters:
        num (int): Integer value to be tested to see if it is prime.

    Return:
        bool: True if 'num' is prime. False otherwise.

    Preconditions:
        num > 1
    """
    i = 2
    while i < num :
        if num % i == 0 :
            return False
        i = i + 1
    return True
```

The input to the function is the number `num`, this is the number that we wish to test if it is prime. Notice in the comment there is a **precondition**. As we discussed earlier, preconditions form a “contract” with the user; the function will work only if certain conditions are met. So, for example, if someone uses `is_prime` with a number less than 2, then the result could be anything - in this case, an incorrect value will be returned, in other cases the function could cause an error - but that is the caller’s problem because this falls outside the contract.

The first line of the function sets `i` to 2, `i` is the counting variable that is being used to keep track of the current value to test. It is common coding practice to use `i` (and `j` and `k` as counting variables in loops). The next line is a while, this while tests if `i < num`, this will mean that we can test the divisibility of all the numbers less than `num`. The first line of the body of code in the while is an if statement. Inside the while loop, all we need to do is check if `i` divides `num` using the `%` operator. If `i` does divide `num` then `num` is not prime, therefore the body of the if statement is to simply return False. Notice that this takes advantage of the way return works: *a return statement will end the function immediately*; at this stage, we already know that `num` is not prime, so we can return immediately and ignore the rest of the function. The last line of the while body is to increment *i* by 1, this moves onto the next number to check. The last line of the function is to return `True`. This again uses the “stop immediately” property of return

statements: if the function has not returned `False` by now, then the `if num % i == 0:` test never became `True` during the while loop, so we know that `num` must be prime, so we return `True`.

Here are a couple of examples of `is_prime`. Try out some more to test your understanding of the function.

```
>>> is_prime(4)
False
>>> is_prime(101)
True
```

This code works fine but we can do better! Firstly note that if 2 does not divide `num` then there is no point testing if other even numbers divide `num` (as all even numbers are divisible by 2). We only need to consider odd `i`. Secondly if `i` does divide `num` then there is a `j` such that `num == i*j`. Therefore, if `num` has a factor then one will be less than or equal to the square root of `num`. Summarising, we only need to test if 2 divides `num` and if any odd `i` less than or equal to the square root of `num` divide `num`.

The function below implements these ideas. Update the definition to match the following code, or download `is_prime2.py`.

```
import math

def is_prime(num) :
    """Returns True iff 'num' is prime.

    Parameters:
        num (int): Integer value to be tested to see if it is prime.

    Return:
        bool: True if 'num' is prime. False otherwise.

    Preconditions:
        num > 1
    """
    if num == 2 :
        return True
    elif n % 2 == 0 :
        return False

    sqrt_num = math.sqrt(num)
    i = 3
    while i <= sqrt_num :
        if num % i == 0 :
            return False
        i = i + 2
    return True
```

The first line introduces an `import` statement. This is used to load a module of other functions and data that could be useful. In this case the "`math`" module is used. This module contains many mathematical functions and numbers (such as pi) not present in the default Python libraries, in our case we are using the square root function (`sqrt`). To see more of what is in the math module, try `>>> help(math) after importing the math module.`

The first if statement in the code checks if `num` is 2 and if so, it is obviously prime so `True` is returned. The elif statement deals with divisibility by 2, if `num` is divisible by 2

then it is not prime, therefore `False` is returned. `sqrt_num` is set to the square toot of `num`. To do this, we call the square root function (`sqrt`) of the `math` library using the syntax `math.sqrt(num)`. `i` is then started at 3 as we are now checking the odd numbers only. The while loop will keep going while `i <= sqrt_num` and will terminate when this test becomes false (i.e. when `i > sqrt_num`). The if statement checks divisibility of `num` with `i` as before and returns `False` if that is the case. `i` is then incremented by 2, moving to the next odd number. As before, if the function has not returned by the end of the while loop, then the last line will be executed returning `True` (i.e. num is prime).

Now that we have updated the function, we can test it again.

```
>>> is_prime(2)
True
>>> is_prime(9)
False
>>> is_prime(19)
True
```

Thinking more about how to test for primality enabled us to write more efficient code. The second version of the `is_prime` function is more efficient as there are fewer iterations through the while loop. This means the code will return an answer faster (especially for larger numbers). Try comparing the two functions with really large numbers and see if there is a difference. Although efficiency is very important in software engineering, we leave more detailed and formal discussions of efficiency to later courses.

The Next Prime

The next function to define is `next_prime` which takes an `num` and returns the next prime after `num`. We will use the same idea as in the previous function - i.e. increment by twos. So we will do slightly different things depending on whether `num` is odd or even.

```
def next_prime(num) :
    """Returns the next prime number after 'num'.

    Parameters:
        num (int): Starting point for the search for the next prime number.

    Return:
        int: The next prime number that can be found after 'num'.

    Preconditions:
        num > 1
    """
    if num % 2 == 0 :
        next_number = num + 1
    else :
        next_number = num + 2

    # next_number is the next odd number after num
    while not is_prime(next_number) :
        next_number = next_number + 2
    return next_number
```

Looking at the code in detail – we start with an if statement that tests if `num` is even. We use the variable `next_number` to be the next candidate for being a prime and we initialise it to be either one or two more than `num`, depending on whether `num` is odd or

even. We have added a comment to remind us about this. The while loop simply increments `next_number` by 2 until it becomes prime. Note that the test in the while loop is the logical negation of the `is_prime` test. In other words, we continue looping while `next_number` is not a prime.

Here are the results of testing.

```
>>> next_prime(3)
5
>>> next_prime(13)
17
>>> next_prime(101)
103
>>> next_prime(2)
3
>>> next_prime(20)
23
```

The n^{th} Prime

Now we bring it all together by writing the top-level function `nth_prime` that returns the n^{th} prime number.

```
def nth_prime(n) :
    """Returns the n'th prime number.

    Parameters:
        n (int): The number of prime numbers to find.

    Return:
        int: The n'th prime.

    Preconditions:
        n > 0
    """
    next_prime_number = 2
    i = 1
    while i < n :
        # loop invariant: next_prime_number is the i'th prime
        i += 1
        next_prime_number = next_prime(next_prime_number)
    return next_prime_number
```

In this example we introduce the idea of a **loop invariant**. A loop invariant is a property that, if it is true at the beginning of the body, then it is also true at the end of the loop body. So if the code in the body satisfies this property and it is also true initially (i.e. when we enter the while loop) then it must be true when we exit the loop. A loop invariant can be useful in helping decide on how variables should be initialised, how the variable should change in the body, and when to exit the loop. It is also helps document what the loop does. So think of the loop invariant and then write the code!

In our example the loop invariant is true initially (because 2 is the first prime). Assuming it is true at the start of the loop, then the code advances `next_prime_number` to the next prime and increments `i` and so the loop invariant is true again at the end of the loop. Therefore, it is also true when we terminate. In which case, not only is the invariant true but also `i == n` and so `next_prime_number` is indeed the n^{th} prime.

By the way, `i += 1` is a shorthand for `i = i + 1`. The file `prime.py` contains all the code above plus a top-level comment (using the triple quotes style). In fact, what we have just done is write our own module! In the interpreter try `import prime` and `help(prime)`, and see how all the trouble we went to writing comments pays off!