Topic 6.1: Functions

When first starting to designing algorithms, one quickly realizes how essential decomposition and abstraction are for managing complexity.

decomposition: breaking a large problem into smaller, independent subproblems

abstraction: hiding the step-by-step details behind a simple, high-level idea so you can concentrate on what something does instead of how it does it.

The primary tool that lets you apply both ideas is the ***function***. A function takes a subproblem, wraps up its logic, and gives it a name. From then on, you can call that function by name whenever you need the solution, treating the inner details as a black box. This lesson will show you how to define your own functions, making your code not just shorter, but far more organized, readable, and easy to build upon.

Defining a Function in Python

To define a function, you use the `def` keyword, followed by the function name, a pair of parentheses `()` , and a colon (`:`). The body of the function is indented.

```
def greet():
    print("Hello, world!")
    print("How are you today?")
```

This defines a function named greet. It doesn't take any inputs and prints a message when called. Defining a function doesn't run its code; you have to call it for it to produce an effect. Calling a function in Python, similar to other languages, is done by writing the name of the function, followed by a pair of parentheses.

```
greet()
```

```
Hello, world!
How are you today?
```

A function can be called many times.

```
for i in range(3):
    greet()
```

```
Hello, world!
How are you today?
Hello, world!
How are you today?
Hello, world!
How are you today?
```

Why use functions?

- Avoid repetition – write once, use many times.
- Break down complex problems – solve one small part at a time.
- Improve readability – a well-named function explains what it does.
- Isolate code – changes inside a function don't accidentally affect the rest of the program (if written correctly).

Parameters and Arguments

Functions become much more powerful when they can accept input. Inside the parentheses you can list parameters variables that the function will receive when called. The actual values passed during a call are arguments.

parameters: variables that the function receives when called

```
def greet(name):
    print("Hello, ", name,"!", sep="")
    print("How are you today?")
```

Now name is a parameter. When we call the function, we supply an argument.

```
greet("Alice")
greet("Bob")
```

```
Hello, Alice!
How are you today?
Hello, Bob!
How are you today?
```

A function can have multiple parameters, separated by commas:

```
def rectangle_area(length, width):
    area = length * width
    print("The area is", area)
```

Call it with two arguments:

```
rectangle_area(5, 3)
```

```
The area is 15
```

The arguments are assigned to parameters by position – the first argument goes to the first parameter, the second to the second, and so on. These are called positional arguments.

Returning Values

All the functions above just printed results. Sometimes you want the function to send back a value so you can use it elsewhere. That's what the return statement does.

```
def square(number):
    return number * number
```

Now calling `square(4)` doesn't print anything; it evaluates to the value `16`. You can store that value in a variable or use it directly:

```
result = square(4)
print(result)
print(square(3) + 1)
```

```
16
10
```

A function always returns a value. If there is no return statement, or if return is used without a value, the function returns the special object `None`. Python's REPL hides `None` when it's the only result, but you can test it:

```
def no_return():
    print("I just print")

value = no_return()
print(value)
```

```
I just print
None
```

Functions that return a value are often called fruitful functions; those that just perform an action (like printing) are sometimes called void functions. In Python, there's really no such thing as a true "void" function, every function returns something, but the concept helps you decide when to use return versus when to just let the function do its job.

A return statement immediately exits the function; any code after it will not run.

```
def is_positive(num):
    if num > 0:
        return True
    return False     # only reached if num <= 0
```

Documentation Strings

It's good practice to describe what your function does. You do this with a *docstring* – a triple-quoted string placed right after the `def` line.

```
def average(a, b):
    """Return the arithmetic mean of two numbers."""
    return (a + b) / 2
```

Docstrings can be accessed with the built-in `help()` function or via the special attribute `.__doc__`.

```
help(average)
```

```
Help on function average in module __main__:

average(a, b)
    Return the arithmetic mean of two numbers.
```

From now on, get into the habit of adding a short docstring to every function you write. It will help you, and others, understand your code quickly.

Exercise