LEEDS
BECKETT
UNIVERSITY

Introduction to Programming

Week 6

Tony Jenkins
a.m.jenkins@leedsbeckett.ac.uk

OBJECTIVE

This week we look at ways to reuse code in a program.

They will enable us to write even more DRY code (and no WET code at all).

Week 6

Pre-requisites

You should have viewed or read the lecture:

"Functions"

on MyBeckett.

You should be able to understand programs that read values and process them.

You should be able to follow a program's *flow*.

You should be starting to be able to write programs that read and validate input, perform nonlinear processing, and produce results.

Functions

Useful Chunks

A function is a piece of tried-and-tested useful code that does something that is needed more than once.

Reusing functions simplifies the development of complex programs.

Working with functions also makes it easier to spread the work around.

Every programming language provides one or more ways to structure code using functions.

Functions may also be called *procedures* or *methods* (although the latter is really something different).

Functions
Useful Chunks

A function is a piece of tried-and-tested useful code that does something that is needed more than once.

We have been using functions all along, because many are built-in to Python.

The `len` function finds the length of, well, anything that has a length.

Note that we have absolutely no idea how it does this, nor do we care. We just know what it does, how to use it, and how to interpret the result.

```
>>> len('spam')
4
>>> len(['eggs', 'spam'])
2
```

```
>>> len(1)
Traceback (most recent call last):
TypeError: object of type 'int' has no len()
```

Functions

Built-In

Many functions like `len` are needed so often that they are built in to the language.

This means that we need do nothing special in order to use them. They are just there.

Other functions are needed often, but not so often. So they are grouped together in *modules*, which can be imported for use when needed.

These modules are part of the standard distribution of Python - they are guaranteed to be available. They are the ***standard library***.

Functions
Modules

Among the most common modules are:

- `string`
- `math`
- `random`
- `statistics`

They provide what it says on the tin.

Docs are online, or use `help` (which is also a function).

Less commonly used modules can still be available as standard - `csv` for example. Others can be downloaded and installed via package managers.

```
>>> import math

>>> help(math)

>>> dir(math)
```

Functions

Modules: Importing

It is possible, but rarely ideal, to import a whole module.

Better is to just import what is needed for the task at hand.

In some cases a function can be imported with an alias (useful to give a name relevant to the current problem or to avoid a namespace clash).

Note that some modules also include useful constants - for example, the `math` module includes a value for Pi.

```
>>> import math
>>> math.sqrt(4)
2.0

>>> from math import sqrt
>>> sqrt(9)
3.0

>>> from random import choice as pick
>>> pick([1, 2, 3])
3
>>> pick([1, 2, 3])
2
```

Functions
Summary

In summary:

- Functions have a name.
- Functions process zero or more values, provided as parameters.
- Parameters are listed in brackets when the function is called.
- Usually, the order of the parameters is important.
- Often, the type of the parameters is important.

Armed with this information, it is reasonable to want to write our own functions.

Functions

Write Your Own

A function definition starts `def` and then includes the statements that make up the function.

In a program, these are not executed when the file is initially interpreted (see where they sit in the file).

If used ("called") in the program, control passes in to the function, and back to the main program once finished.

```
def hello():
    print('Hello, World')


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

Functions

Write Your Own

A function definition starts `def` and then includes the statements that make up the function.

In a program, these are not executed when the file is initially interpreted (see where they sit in the file).

If used ("called") in the program, control passes in to the function, and back to the main program once finished.

This line marks the start of the statements that are executed when the script is run.

```
def hello():
    print('Hello, World')


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

Functions

Write Your Own

A function definition starts `def` and then includes the statements that make up the function.

In a program, these are not executed when the file is initially interpreted (see where they sit in the file).

If used ("called") in the program, control passes in to the function, and back to the main program once finished.

And obviously a function can be used more than once.

```
def hello():
    print('Hello, World')


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

Functions

Write Your Own

A function definition starts `def` and then includes the statements that make up the function.

Parameters, if used, need to match. The value in the calling program is passed into the function.

The value passed in can be in a variable, or can be a literal value.

If in a variable, the identifiers do not have to match (in fact, it's usually better if they don't).

```
def hello(name):
    print(f'Hello, {name}!')


if __name__ == '__main__':
    king = 'Arthur'
    hello(king)

    hello('Galahad')
```

Functions
Returns

A function definition starts def and then includes the statements that make up the function.

Often a function calculates a new and useful value, and sends it back to the calling program.

For example, it is often useful to have a function that removes the last character from a string.

Some languages have such a function, Python does not. So we roll our own.

```
def chomp(s):
    return s[:-1]


if __name__ == '__main__':

    knight = 'Robin\n'
    knight = chomp(knight)

    print(f'Bold Sir {knight}')
```

Functions
And Types

Some languages care very much about the types that can be processed by a function.

Python cares much less (remember that `len` works with anything that has a length).

So Python functions can behave in different ways if different types are passed in.

Usually a function would throw an error if the type(s) provided made no sense.

Functions

Why?

It may seem at first that using functions just complicates matters.

But this is not so.

Working with functions supports abstraction, and lets us work with small chunks of code that can be used again and again.

Ideally, we never want the "code unit" we are working on to be more than about 20 lines long. Functions let us do that.

Functions

Why?

It may seem at first that using functions just complicates matters.

Why 20 lines?

Historically, because this was the amount of code that could be viewed on an old-style terminal without having to scroll.

Practically, because this is about the maximum amount of code that a developer can keep in their head at any point in time.

Functions

Why?

A common "newbie" error is to write a program, intending to "turn it into functions" later.

Don't do this!

This approach just serves to complicate the development and, usually, produces a horrible mess of code.

Much better is to identify functions that will be useful, code and test them, and then bring them together to form the complete program.

Functions

Example

We have before used some built-in functions that work on lists:

- `max`, `min`, `sum`, `len`.

We found that there was one for the average, provided in the `statistics` module.

Now we can craft a function that finds the *range* of a list of numbers - the difference between the highest and lowest.

What should it do if the list provided is empty? Or has just one element? Or if the list contains something that makes the "range" a useless concept?

Program
Exam Marks

A student studies exams, graded 0 to 100.

Write a program that reads these marks, and outputs the highest, lowest, and mean.

The list of marks ends when the user just presses Enter.

Display also the grade, where A is over 70, B over 60, C over 50, D over 40, and F otherwise.

This program now illustrates all the Python we have done so far.

From now on, we need to practice.

Program
Password Checker

A certain University has a rule that all passwords used by its IT users must:

- Be between 8 and 12 character long.
- Contain at least one uppercase letter.
- Contain at least one lowercase letter.
- Contain at least one digit.
- Contain at least one "special character".

Write a program that accepts a string as input and displays whether or not it fits these rules.

```
Password? Password1!
Password Check Result: Check OK.

Password? cheese
Password Check Result: Check Failed.

Password? bhH768jgh
Password Check Result: Check Failed.

Password? yellow9toaster%herring
Password Check Result: Check Failed.
```

NEXT

It is now time to practice, and practice.

And then practice some more.

Thank you

LEEDS
BECKETT
UNIVERSITY