# 1811/2807/7001ICT
# Programming Principles

School of Information and Communication Technology
Griffith University

Trimester 1, 2024

# 15 Using Library Functions

In this section we unlock the standard library and learn to use both built-in functions and the methods of classes.

## 15.1 The standard library

We have covered much of the syntax for Python statements that actually do things.

To go further, we need to explore more of the standard library.

There are many kinds of things to be found in the standard library:

- built-in functions, e.g. `abs()`;
- built-in constants, e.g, `True`;
- built-in types (classes), e.g. `int`;
- constructors of the built-in classes, e.g. `int()`;
- methods of the built-in classes, e.g. `str.isspace()`; and
- functions, constants, and classes imported from modules.

### 15.1.1 Reading documentation

You should not be treating these lecture notes as your sole source of information.

The on-line Python documentation is the primary source for Python programmers.

Our job is to introduce you to it and enable you to use it.

### 15.1.2 Built-in functions

These are functions that are always available for use without either importing modules or invoking as methods.

We will address these other kinds shortly.

There is a relatively short list of built-in functions.

They include:

- the constructors for the types we know `int`, `float`, `bool`, and `string`;
- the `print` and `input` functions we use regularly; and
- other utilities.

An easy one to understand that you should know about now is `abs`.

Look it up now, and read its documentation.

## 15.2 Function calls

A *call* to a function always involves its name and a pair of parentheses.

The parentheses enclose zero or more *arguments* (also known as *parameters*).

Lets call a couple of functions in the REPL.

```
>>> print(-42)
-42
>>> abs(-42)
42
>>>
```

Both produce output, but what happens when these statements are in a script?

```
# script: printAbs1.py

print(-42)
abs(42)
```

```
$ python3 printAbs1.py
-42
$
```

This time only the call to `print` produced output.

`print` and `abs` work in quite different ways.

`print` actually prints.

`abs` *returns* a value. We only saw that value in the REPL, because the P in REPL is for "print".

### 15.2.1 Returned values

In general, functions receive values to work with via their arguments, and they return a value when they exit.

Some functions may also cause side-effects, such as printing.

To see the value returned by `abs` in a script, we have to print it.

```
# script: printAbs2.py

print(-42)
print(abs(42))
```

```
$ python3 printAbs2.py
-42
42
$
```

### 15.2.2 None

If all functions return a value, what does `print` return?

Let's print the value returned by `print`.

```
>>> print(print(-42))
-42
None
>>>
```

The `-42` was printed by the inner `print` call.

Its returned value, `None`, was printed by the outer `print` call. `None` is a special value in Python that means "there is no information here".

In other languages, such as C or Java, no information is represented by `void`.

### 15.2.3 Order of arguments is significant

When functions have more than one argument, the order of the arguments is very important.

For example, the constructor for a `range` can have 1 to 3 arguments, the meaning of each one is different.

```
>>> for i in range(1, 3): print(i)
...
1
2
>>> for i in range(3, 1): print(i)
...
>>>
```

(A blank input to the prompt `...` ends the body of the `for` loops.)

### 15.2.4 keyworded parameters and default values

After the ordinary parameters, where the order is significant, a call's parameter list may include optional arguments, that are named with a keyword and have default values (which is how they can be optional).

We have seen this with, for example, `print`.

With the keyworded arguments `sep` and `end` we can control the format of the output.

```
>>> print(1, 2, 3)
1 2 3
>>> print(1, 2, 3, sep = ', ', end = '.\n')
1, 2, 3.
>>>
```

## 15.3 Calling a method

Time to learn a little more object-oriented programming.

This is our definition of a class, so far.

> *Definition*: A class is a data type that defines the properties for any objects that are instances of this class.

Now we will refine and extend this a bit.

> *Definition*: A class is a data type that defines the *attributes* (or properties, or fields) and *methods* for any objects that are instances of this class.

*Definition*: An attribute of a class is an item of data stored in the instances (objects) of a class.

*Definition*: A method of a class is a function that defines operations that can be performed on and/or with the instances of a class.

*Definition*: Together, the attributes and methods of a class are called its *members*.

We know that instances of a class, such as `str`, store information inside them.

That information will be stored in the attributes of the objects.

Now we know that the class also defines methods that can operate on these objects.

If we look in the standard library, below the built-in functions, we find the built-in types (classes).

This documents everything we need to know about Python's built-in classes.

It will not usually document the attributes as that information is usually kept private, but it does document the methods we can use with each class.

See for example the list of String methods.

### 15.3.1 str methods

This is not the place to replicate the entire python library documentation, so here are links to the most useful `str` methods, categorised.

**case conversions:** `capitalize`, `lower`, `upper`.

**removing whitespace:** `lstrip`, `rstrip`, `strip`.

**formatting:** `center`, `format`, `ljust`, `rjust`.

**testing:** `endswith`, `isalnum`, `isalpha`, `isdigit`, `islower`, `isspace`, `isupper`, `startswith`.

**searching:** `count`, `find`, `index`, `rfind`, `rindex`.

**parsing:** `partition`, `rpartition`, `split`, `splitlines`.

**modifying:** `replace`, `translate`.

### 15.3.2 Using a method

The way that we use a method that is different to an ordinary function may be demonstrated in the REPL. For example:

- `len` is an ordinary function built-in that takes a string as an argument.

```
>>> len("abc")
3
>>>
```

  The string is passed as an ordinary argument.

- `isdigit` is a method of class `str` and is called like this.

```
>>> "abc".isdigit()
False
>>> "123".isdigit()
True
>>> s = "42"
>>> s.isdigit()
True
>>>
```

  Methods must be called using an object (in the example, the strings) and the member access operator, a period (`.`).

  The object is passed to the method as an extra argument.

  In this case the parameter list is empty, but there could be other parameters.

### 15.3.3 str.format

One string method that is particularly useful is `str.format`.

Among other things, it can help us control the number of decimal places when we print `floats`.

```
>>> dollars = 10.0 / 3
>>> dollars
3.3333333333333335
>>> print(dollars)
3.3333333333333335
>>> print("dollars = ${:.2f}".format(dollars))
dollars = $3.33
>>>
```

The string defines the format of the output.

The arguments of the `format` method are interpolated into the output string.

The braces indicate where the values will be interpolated.

The special sequence of characters inside the braces control the conversion of the value to a string, in this case, that it is a float value to be converted with two decimal places.

The full documentation for how to create format strings is here.

Here are a few more examples.

- Interpolating multiple values, in order, with default formatting.

```
>>> "{} / {} = {}.".format(10, 3, 10 / 3)
'10 / 3 = 3.3333333333333335.'
>>>
```

- Using the values out of order.

```
>>> "{2} {1} {0}".format(10, 20, 30)
'30 20 10'
>>>
```

- Using the values out of order, and formatting them in decimal or hexadecimal.

```
>>> "{2:d} {1:d} {0:d}".format(10, 20, 30)
'30 20 10'
>>> "{2:X} {1:X} {0:X}".format(10, 20, 30)
'1E 14 A'
>>>
```

- Using the values in order, and formatting them in in scientific notation with 3 decimal places.

```
>>> "{:.3e} {:.3e}".format(2 ** 20, 2 ** 40)
'1.049e+06 1.100e+12'
>>>
```

- Using the values in order, and formatting them in in scientific notation with 3 decimal places, in field widths of 12 spaces, right justified.

```
>>> "{:12.3e} {:12.3e}".format(2 ** 20, 2 ** 40)
'   1.049e+06    1.100e+12'
>>>
```

- Formatting strings in fixed field widths, left justified.

```
>>> "{:<10s} {:<20s}".format("name:", "Fred")
'name:      Fred                 '
>>>
```

By controlling the field widths, multiple values may be output in tables with neat columns.

Numbers should be output right justified, and strings left justified.

## 15.4 Importing library modules

Many useful things in the standard library are not built-in, that is not immediately and always available.

The extras are declared in modules which must be imported by your script.

An example module is the `math` module.

There are two forms of the `import` statement.

This one

```
import  module-name
```

imports the definitions in the named module, but they must be used *qualified*, by the module name.

```
# script: ceilFloor1.py
# use ceil and floor from the math module qualified

import math

x = 10 / 3
print("x = {:.4f}".format(x))
print("ceil(x) = {:.4f}".format(math.ceil(x)))
print("floor(x) = {:.4f}".format(math.floor(x)))
```

```
$ python3 ceilFloor1.py
x = 3.3333
ceil(x) = 4.0000
floor(x) = 3.0000
$
```

The other form of import statement, starting with `from`

`from` *`module-name`* `import` *`name(s)`*

imports only the comma-separated *`name(s)`* from the named module, and they may now be used unqualified.

```
# script: ceilFloor2.py
# use ceil and floor from the math module unqualified

from math import ceil, floor

x = 10 / 3
print("x = {:.4f}".format(x))
print("ceil(x) = {:.4f}".format(ceil(x)))
print("floor(x) = {:.4f}".format(floor(x)))
```

This variation

```
from module-name import *
```

imports all of the top-level definitions from the named module and allows their use unqualified.

## Section summary

This section covered:

- parts of the standard library beyond the built-in functions:
  - the methods of the standard classes; and
  - functions imported from modules;
- how to call functions and methods with arguments;
- how some functions return values and others return `None`; and
- some useful string methods, including `format`.