Fundamentals of Computer Programming

Lecture slides - Scripts and Modules

Intro

- This lesson covers
  - Placing code in text files
  - Executing scripts
    - Accessing command line arguments
  - Writing modules
    - Methods of importing
    - The module search path

Using program files

- Working in *interactive mode* (i.e. typing directly into the Python interpreter) is good for learning, experimenting and writing small programs (*snippets*)
- However, when writing more substantial programs we need to store our program code (called *scripts)* in one or more text files
- These are plain text files that must contain valid Python statements and definitions, and be named with a `.py` suffix
- We then use these files as input into the Python interpreter allowing us the “run” or “execute” the *scripts*
- We can also define *functions* and *classes* within our files, that may be reused across several different programs

Regular text files with valid Python commands. Has to have a .py extension

Writing and Executing Scripts

- We can write the code using any generic text editor or by using an Integrated Development Environment (**IDE**) - We use Pycharm
- Once saved, we can execute *scripts* stored within a file from a command line by directly passing the filename to a Python interpreter, e.g.

```
python my_program.py
```

...output of program execution shown here

- *Note: interactive mode uses a **REPL** approach, so output of evaluations is done automatically without the need for a print() statement, this is not true when executing scripts*

Accessing command line arguments

- When a *script* is executed by the Python interpreter, it is possible to pass *command line arguments*
- These allow the user to pass in useful information to the program, e.g. we could pass values to a program that calculates an average

```
python average.py 10.2 8.8 2.6
```

```
Average is 7.2
```

Useful because you can pass information into the program
↳ Makes program code more flexible

- The arguments are accessed within the program from **`sys.argv`**, which is a *list* of *string* values
- The first element of **`sys.argv`** is the program name, such as '`average.py`', and the remaining elements are the passed values

↳ List of anything in the command line after program name. ALWAYS strings

Example: average.py script

useful code to make note of

```
# A Python script file 'average.py' that calculates an average

import sys

values = sys.argv[1:]     # splice args list to remove the file name

total = 0;

for n in values:          # total the remaining values

    total += float(n)         # convert each string 'n' to a float


# calculate and print the average

print("Average is", total/len(values))
```

→ Removes the filename from the list and stores them in the variable 'values'

Writing Modules

- As an alternative to executing code as a *script*, we can **`import`** a file as a *module* into other programs
- Any **`.py`** file containing *definitions* and *statements* is called a **module**
- We have already seen the use of **`import`** to access modules that are built-in to Python, e.g.

```
import sys          # import the sys module
```

- We can also import our own *modules* in the same way, by using the file name without the `.py` suffix
- When a *module* is imported any Python statements are executed, these are usually used to *initialise* the module e.g. setup variable values etc.

Example: utils.py module

This is a module over a script because it is defining a function

```
# A Python module file 'utils.py' that defines an average() function
def average(values):
    """ Calculates the average of the given list. """
    total = 0;
    for n in values:          # total the given values
        total += float(n)
    return total/len(values)  # return calculated average

# initialisation statement
print("Welcome, utils module has been imported and initialised!")
```

Example: Importing the ‘utils’ module

- We can now *import* the ‘utils’ module and call its functions from within other Python programs, or from directly within the interpreter, e.g.

```
import utils      # import a module from the ‘utils.py’ file
```

```
Welcome, utils module has been imported and initialised!
```

Modules are so useful because it allows you to write code once and then share it over multiple projects.

```
# call the imported function a few times
print("Average is", utils.average([10, 23, 30]))
```

```
Average is 21
```

```
print("Another average is", utils.average([10.2, 8.8, 2.6]))
```

```
Another average is 7.2
```

Methods of importing

- Imported modules have their own *symbol table* that means the variables they define do not interfere with those in other modules or the *main* module
- Modules can be imported in different ways, which impacts how they are used within the program, and how the local *symbol table* is effected
- Importing a whole module allows access to all definitions included in that module, but accesses must be *prefixed* with the module name, e.g.

Where Python stores a list of where variables + functions are available

```
import math                    # import the whole math module

print("Square root of x is", math.sqrt(x)) # must prefix function call
```

Doing it this way stops potential conflict with variables I write.

- In this example only the variable '`math`' is created in the local *symbol table*

Methods of importing

- An import can also include an *alias*, which renames the module within the local *symbol table* e.g.

```
import math as m      # import the whole math module, but as 'm'

print("Square root of x is", m.sqrt(x))    # now can prefix with 'm'
```

Allows import with alias instead of full variable

- This avoids name clashes e.g. your program may already have a variable or function called '`math`' that you did not want to overwrite
- As an alternative, specific elements can be directly imported into the local *symbol table*, e.g.

```
from math import sqrt, pi      # directly import sqrt() and pi

print("Square root of x is", sqrt(x))          # no prefix required
```

Methods of importing

- Directly imported elements can also use aliases, e.g.

```
from math import sqrt as root

print("Square root of x is", root(x)) # alias name used
```

- It is also possible to import **all** of the content of a module into the local *symbol table*, using a wildcard character (*), e.g.

```
from math import *
```

Creates EVERYTHING in the local symbol table which can cause conflicts

- This is not recommended however, since there is high chance that clashes between imported and existing variable names will occur
- Only use this form when working with the interpreter in interactive mode

The Module Search Path

- When an `import` statement is executed, a check is first made for a built-in module that has the given name
- If the name is not known, then a list of directories stored in the **`sys.path`** variable is searched for the file to be loaded

this variable is defined automatically - list of directories on the local machine.

- The **`sys.path`** list is initialised when the program starts, and includes the directory from which the input script was loaded, or the current directory if no script was specified
- It also includes the directories specified by the `PYTHONPATH` environment variable (set within the host OS)

Operating System sets. Exist in most operating systems.

- The **`sys.path`** list can also be changed by the program itself, to ensure a certain location is included in the search path

Change it to where you know your modules are stored

Listing imported names

- When working in interactive mode (i.e. typing directly into the Python interpreter) it is useful to find information about imported modules
- The *built-in* **`dir()`** function lists information about an object, including imported modules

lists information about things in a module. Functions and variables defined within a module

- If called without an argument the function lists all known names within the local *symbol table*
- If given an imported module name it will list all names (i.e. functions, types and variables) defined within that module
- Note: the `dir()` function is only really of use in interactive mode, and may change behaviour in future versions

Example: dir() function

```
import math
dir(math)
```

This command is showing all the functions and variables defined in the math module.

Only really useful in interactive mode

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum',
'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf',
'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh',
'tau', 'trunc']
```

```
type(math.sqrt)       # check the data-type of the math.sqrt variable
```

```
<class 'builtin_function_or_method'>
```

Using the module ‘name’

Important to do on EVERY program

Every module or variable is required to have one

- When a program executes, it can access its own ‘name’
- This is done using the special variable `__name__`, which represents the current *module* name
- This is useful since it allows a program to determine whether it is being executed as a *script*, or imported as a *module*
- A program is running as a *script* has a module name of ‘`__main__`’ rather than the name of the `.py` file
- Hence, a program can use this information to be flexible enough to be used as either an executable *script* or an as imported *module*

Example: A ‘script’ and ‘module’

```
# A program that can execute as a ‘script’ or be imported as a
‘module’

def average(values):

    """ Calculates the average of the given list. """
    total = 0;
    for n in values:            # total the given values
        total += float(n)
    return total/len(values)  # return calculated average


if __name__ == "__main__":    # check if executed as a script
    import sys                # if so call the function with the args
    print("Average is", average(sys.argv[1:]))
```

Allows variables to be a script or a module

When this is set it knows to run it as a script.

Will allow this code to run
↳ Can pass command line arguments to the average function defined inside this script

Summary

- Python program files (`*.py`) can be *executed* as **scripts** or *imported* as **modules**, or sometimes both
- We can access the arguments passed when a *script* is *executed* via the **`sys.argv`** list type value
- A set of built-in *modules* are provided, but we can also write our own
- Variations of the `import` statement exist that determine how module names are imported into the local *symbol table*
- When an `import` statement is executed, a *module search path* is used to find the file to be imported
- Using the `__name__` variable allows development of *flexible* programs