MODULAR PROGRAMMING

DR IAN CORNELIUS

HELLO

- Learning Objectives
    1. Understand the concept of modular programming and its purposes in Python
    2. Demonstrate your knowledge of using modules

MODULAR PROGRAMMING

- Breaking down large programming tasks into smaller tasks or **modules**
- Individual modules can build up to create a larger application
- **Advantages**:
  - simplicity
    - modules focus on a small portion of a given problem
  - maintainability
    - logical boundaries are enforced between the different problem domains
    - minimises interdependency between code making it more maintainable
  - re-usability
    - functions in a single module can be re-used through an interface

OVERVIEW OF MODULES IN PYTHON

- Three methods of defining a module:
  - writing a module in the Python language
  - writing a module in C and loading it dynamically
  - using a built-in module, contained in the Python interpreter
- Each modules content is accessed the same way, using the `import` keyword
- A module is the same a code library
  - files containing classes and functions that can be imported

CREATING A PYTHON MODULE

- Create an ordinary Python file (“`.py`”) with some code
  - i.e. a file called `calculator.py`

```
class Calculator:
    def __init__(self):
        pass
    def cube(self, x):
        return x * x * x
    def square(self, x):
        return x * x
```

- These files may be stored in a separate directory
  - i.e. in my example they are stored in `modules`
- Modules stored in a separate directory need an `__init__.py` file
  - assists the Python interpreter in locating the submodules
    - i.e. `calculator.py`

```
from . import calculator
```

USING A MODULE

- A module can be used by importing the filename
  - i.e. `calculator.py` can be imported by using:

```
from modules.calculator import Calculator
```

- Functions can then be accessed from the `Calculator` class

```
calc = Calculator()
calc.cube(3)
calc.square(3)
```

MODULE ALIASES

- Aliases can be provided to a module when it is imported using the `as` keyword
- Useful for shortening a module name
    - i.e. `Calculator` could become `Calc`

```
from modules.calculator import Calculator as Calc
```

```
myCalc = Calc()
myCalc.cube(3)
myCalc.square(3)
```

EXECUTING MODULES AS A SCRIPT

- Modules in Python can be executed as script in the command-line as such:

```
python3 filename.py < arguments >
```

- To read arguments from the command-line, you need to use the `sys` module
    - additionally, you need to read the input from the commandline using the `argv` method

```
if __name__ == "__main__":
        import sys
        from modules.calculator import Calculator
        print(f"Cube of {int(sys.argv[1])} = {Calculator().cube(int(sys.argv[1]))}")
```

```
$ python3 math.py 2
  Cube of 2 = 8
```

BUILT-IN PYTHON MODULES

- Python consists of several built-in modules
  - i.e. `platform`, `os` and `sys`
- Contains resources for specific functionalities
  - i.e. operating system management, file handling, network and database connectivity
- Additional Resource:
  - List of Built-In Modules

```
import platform
```

```
systemDetails = platform.system()
```

```
systemDetails = Windows
```

LISTING ALL FUNCTIONS IN A MODULE

- To determine all the functions or variable names in a module, the `dir()` function can be used

```
import platform
```

```
moduleDetails = dir(platform)
```

```
moduleDetails = ['_Processor', '_WIN32_CLIENT_RELEASES', '_WIN32_SERVER_RELEASES', 'builtins', 'cached', 'copyright',
'doc', 'file', 'loader', 'name', 'package', 'spec', 'version', '_comparable_version', '_component_re',
'_default_architecture', '_follow_symlinks', '_get_machine_win32', '_ironpython26_sys_version_parser',
'_ironpython_sys_version_parser', '_java_getprop', '_libc_search', '_mac_ver_xml', '_node', '_norm_version',
'_os_release_cache', '_os_release_candidates', '_os_release_line', '_os_release_unescape', '_parse_os_release',
'_platform', '_platform_cache', '_pypy_sys_version_parser', '_sys_version', '_sys_version_cache', '_sys_version_parser',
'_syscmd_file', '_syscmd_ver', '_uname_cache', '_unknown_as_blank', '_ver_output', '_ver_stages', 'architecture',
'collections', 'freedesktop_os_release', 'functools', 'itertools', 'java_ver', 'libc_ver', 'mac_ver', 'machine', 'node',
'os', 'platform', 'processor', 'python_branch', 'python_build', 'python_compiler', 'python_implementation',
'python_revision', 'python_version', 'python_version_tuple', 're', 'release', 'subprocess', 'sys', 'system',
'system_alias', 'uname', 'uname_result', 'version', 'win32_edition', 'win32_is_iot', 'win32_ver']
```

- **Note**, the `dir()` function can be used on all modules
    - you can also use `help()` to find out information about the module

GOODBYE

- Questions?
  - Post them in the **Community Page** on Aula
- Contact Details:
  - Dr Ian Cornelius, ab6459@coventry.ac.uk