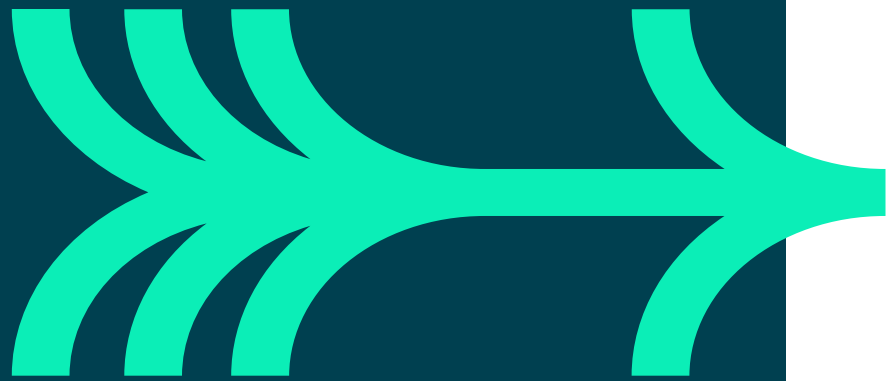
Importing code packages, modules, and namespaces



The Topic: What?

•Taking code re-use up a level

- Modularity
- Building a Package
- Is it a program or a module?
- Namespaces
- The __main__ namespace trick
- Unit Testing

Applications: Why?

•To be able to break down a complex problem into more manageable sub-tasks.

•To create code that can be re-used, that is simple, and easily maintained.

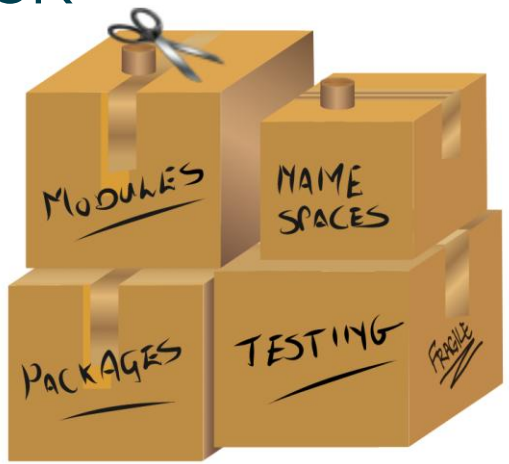
Expectations: Who?

•Learners are expected to have covered tuples, lists, dictionaries, and sets in Python previously.

Packages, modules, namespaces

Let's take CODE REUSE up a level!

- Modularity
- Building a Package
- Is it a program or a module?
- Namespaces – one honking great idea!
- The_main_namespace trick
- Unit Testing



Modular programming

Modular programming is the process of breaking down a large complex problem into separate, smaller and more manageable subtasks or modules. These simpler individual modules can then be completed (possibly in parallel for larger teams) and then glued back together like building blocks.

There are several advantages to using a modularised approach including:

- Simplicity: a module typically focuses on one small part of the problem and is easier to design and code.

- Maintainability: module are independent and designed, coded, debugged and tested and modified with impacting others.

- Reusability: modules can be reused across projects and programs. The 'Holy Grail'!

- Scoping: modules can have a separate namespace (honking great idea) avoiding conflicts with identifier names.

Modular programming

Python supports Packages, Modules, and Functions that all promote the concept of modularisation.

There are three ways to define a module to be used in Python:

1. It can be written in Python itself with a .py suffix. This is the most common and the one we will focus on.

2. It can be written in C and loaded dynamically at run-time. The re module uses this!

3. A built-in module is intrinsically contained with the compiler, e.g., built-ins.

Python packages

Python Package

Technically, a Python Package is just another module with sub-modules, sub-packages and a __path__ attribute. So importing a module or package is really just the same. But if you are looking for a definition - then a regular Package is NAMED directory with a logical group of files, modules, sub-folders AND a file called __init__.py. If the __init__.py is missing then its called a Namespace package.

In general, sub-modules and sub-packages are not imported when a package is imported. The __init__.py can be used to include all or any sub-modules or sub-packages. It can also contain global constants and functions and can also be empty!

Python automatically stores compiled python modules in a __pycache__ folder which is found in the Package directory or Python install directory. When you are importing a module, including Python Standard Library modules, it is the pre-compiled modules with a suffix .pyc which are loaded, reducing the time to compile. If the .py module file has a newer timestamp then it is re-compiled and copied back in to the __pycache__ folder.

Finding modules

The Python interpreter finds the modules - as a built-in, in the current or package directory, in a list of directories defined in the PYTHONPATH environment variable, or an installation-dependent list of directories configured during Python install.

On Linux, $ export PYTHONPATH="${PYTHONPATH}:${HOME}/python/lib:/projectX/lib"

On Windows, [Right Click]Computer > Properties >Advanced System Settings > Environment Variables >System Variables >New

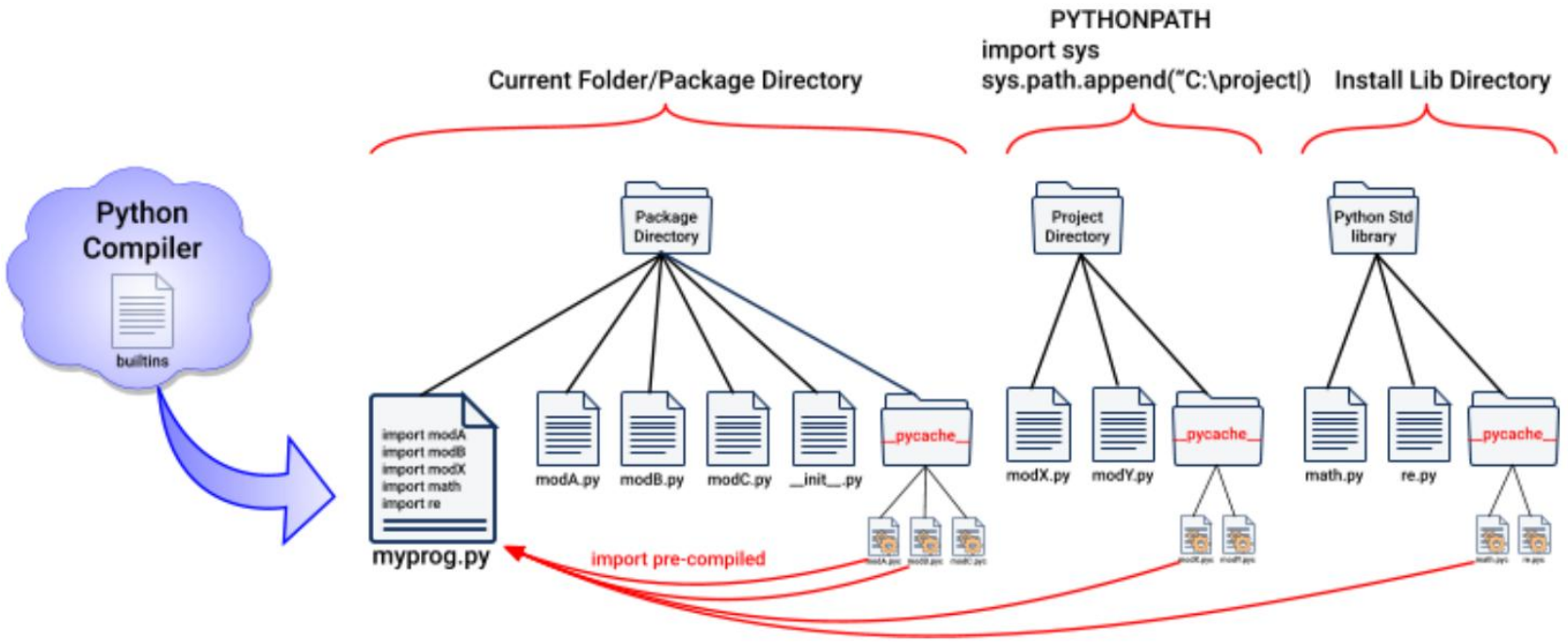
Within a program,

```
>>> import sys
>>> sys.path.append(r"C:\labs\projects\Proj_X")
>>> print(sys.path)
```

Importing a Package, and then a sub-package (from the current package),

```
>>> import my_package
```

Finding modules

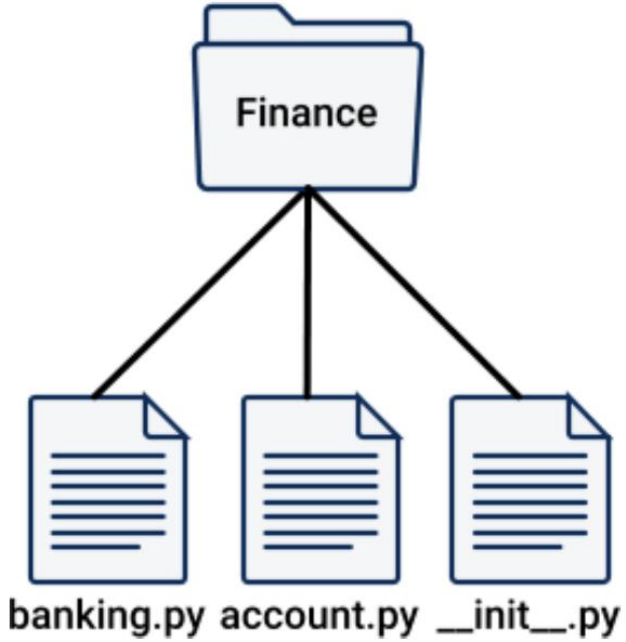


Creating a package

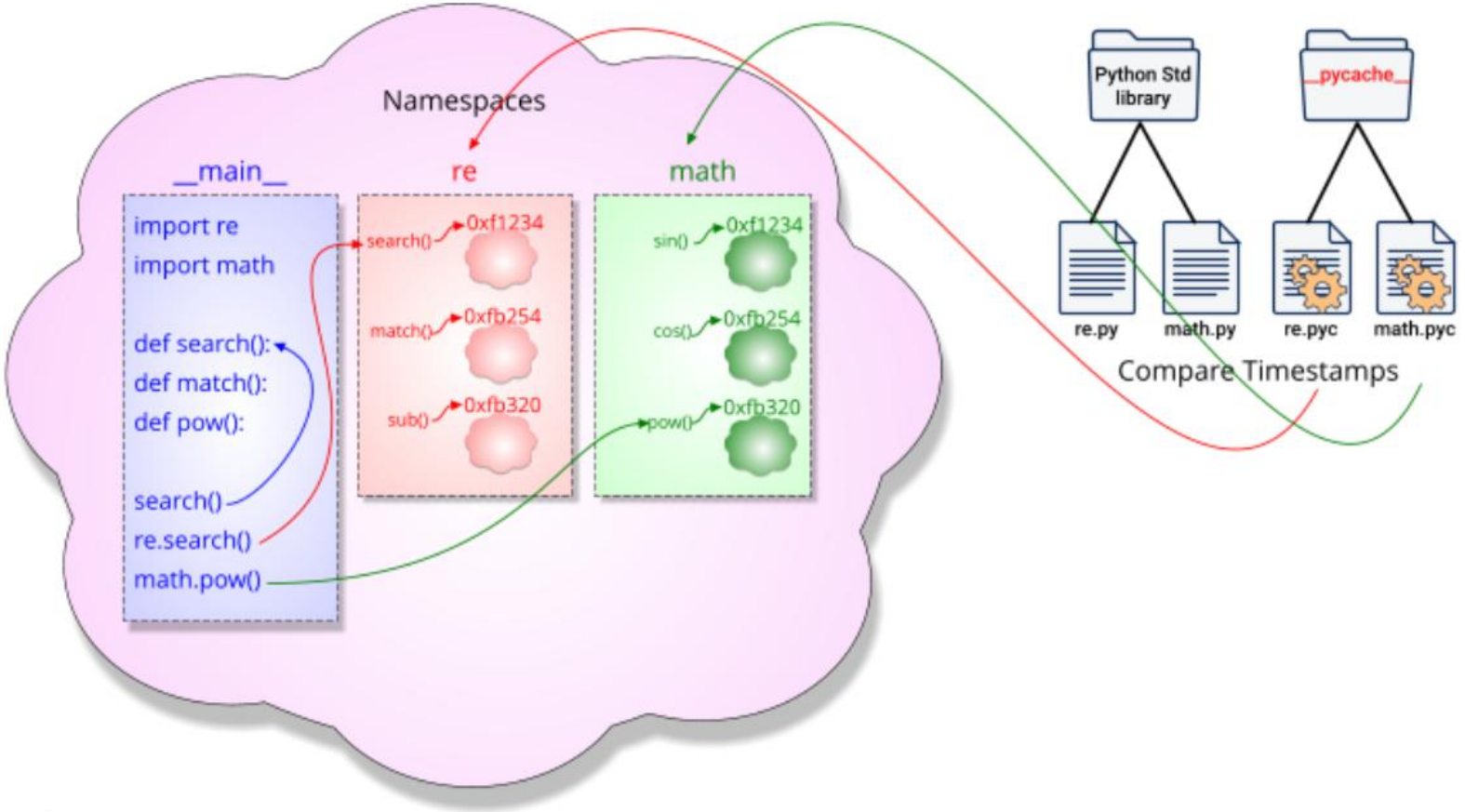
Trainer demonstration

banking.py

Packages are a logical group of modules that share the same directory and namespace and help avoid naming collisions using the **dot notation** - in a similar way that hierarchical file systems can have files with the same names but with a different pathname. Packages can be created and named in Pycharm, and a new directory will be created with the __init__.py.



Packages and namespaces

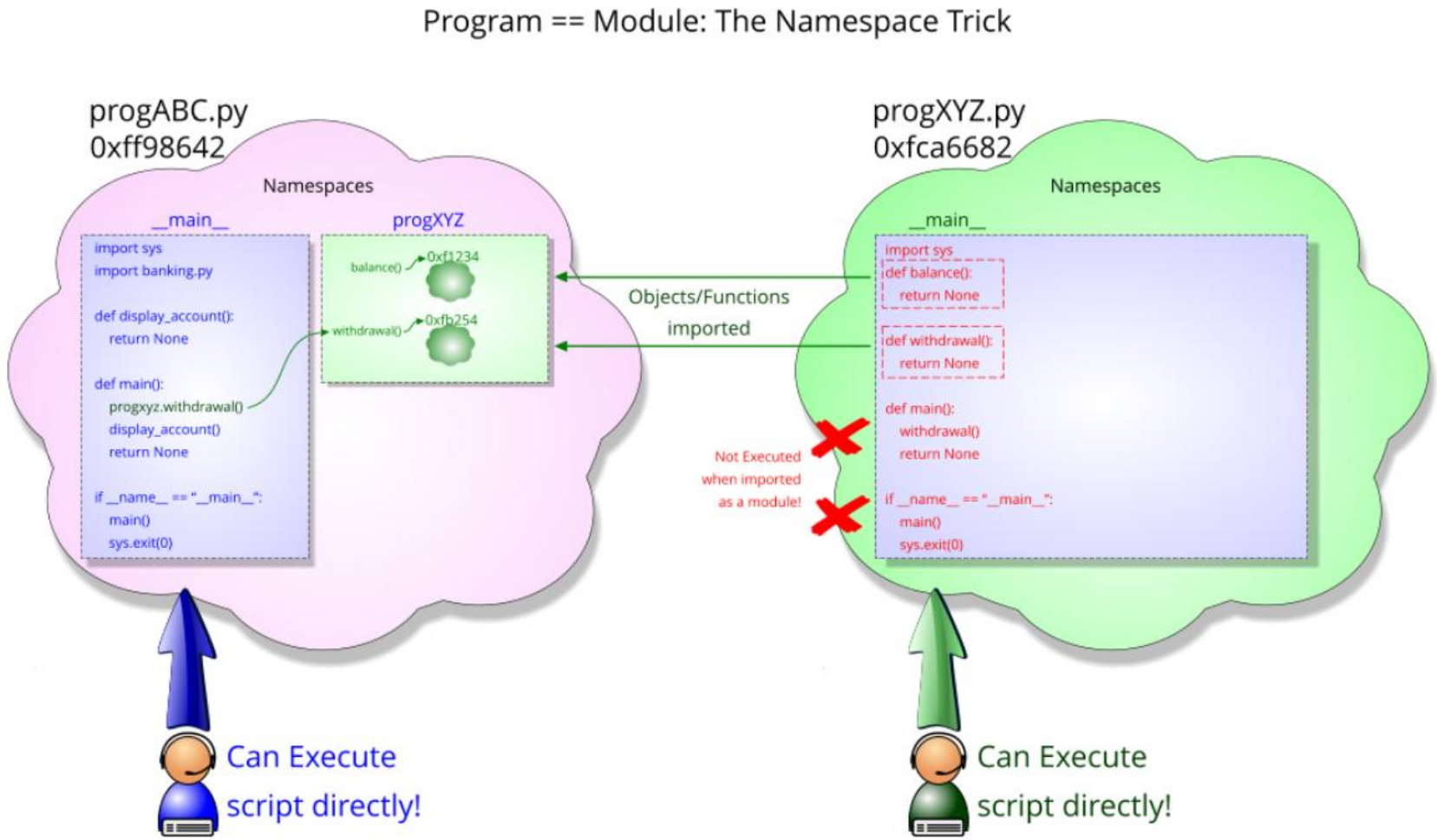


Packages and namespaces

Trainer demonstration

account.py

Namespace trick



Namespace trick

Have you noticed the __name__ == "__main__" test we have been using in conjunction with a main() function? It's called the namespace trick and allows your script to act as a module and your module to act as a script. In other words, it allows the user to execute the module directly as a script, and also import the module into another script without executing the main() function!

It's also part of structuring our python scripts correctly. Continue using it! Oh, and always end your script with an error code, 0 indicates zero errors and any other integer (1-255) indicates an error code! This exit code is passed back to the calling program and indicates success or failure of your script.

Namespace trick

Importing – the different ways

import bank_account bank_account.deposit()	**Safer and traceable**
import sys, re, bank_account	**PEP8 disapproves!**
import sys import re import bank_account	**PEP8 approves!**
import bank_account as b_acc b_acc.deposit()	**Module alias**
from bank_account import *	**Namespace Collision**
from bank_account import deposit	**Safer, be careful!**
from bank_account import (deposit as dp) dp()	**Confusing**

Unit testing

Unit testing your code is an essential part of the software development life cycle. As code is written or modified, it must be tested before deployment and existing code must be retested (regression testing) to ensure that the software still behaves as expected (i.e., no unwanted side effects).

Most programmers have done some form of testing, with exploratory testing the most natural. Exploring the code to find bugs, issues, invalid inputs but without a plan. Manual testing is more formal, as it incorporates a list of all the features to test, different inputs and expected results. And every time a change is made, we have to go manually go through the list again. And the fun ends!

Automated testing is more thorough and dependable with far better results. In this case, the plan of tests is scripted. Python comes with a set of tools and libraries to help create automated tests. Testing is a huge topic in programming and Python but we will learn to create and execute a basic test using the Python Standard Library unittest module.

Unit testing

Unit testing, as the name implies is about a smaller test that checks a single component. If you are testing multiple components then that is integration testing. In unit testing, when we are calling a single function with parameters, this is called a test step. And when we are checking the results, this is called a test assertion. Python already has an assert() function.

>>> assert sum([10, 20, 50]) == 80, "Should be 80" # Returns an AssertionError and "Should be 80" if fails.

>>> assert sum((1, 1, 1]) == 3, "Should be 3" # Returns an AssertionError and "Should be 80" if fails.

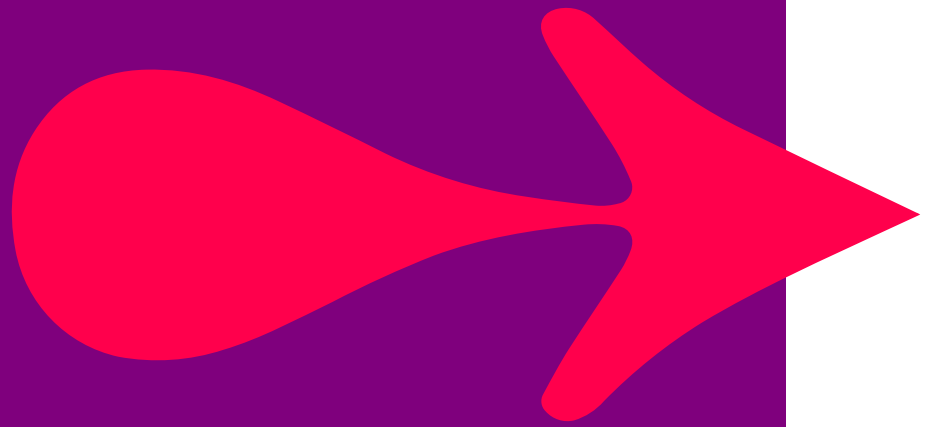
This can then be incorporated in function code:

```
from demo_calculator import add, multiply

def test_sum():
    assert add(10, 20) == 30, "Should be 30"

def test_multiply():

assert multiply(10, 20) == 200, "Should be 200"

if __name__ == "__main__":
     test_sum()

test_multiply()
```

This is ok for simple checks, but for multiple failures we need a test runner which is a special app designed for running tests and validating output etc. The unittest module has both a test framework and runner. The initial difference is we put our tests into classes as methods (which we introduce in the next session!) and use special assertion methods from the module rather than the built-in assert function.

Unit testing

Trainer demonstration



demo_unittest_calc.py

Learning check

5-10 mins

Quiz!

1. **Name the command used to load a Python module?**
2. **Fill in the blank.**

   Python modules have a ____ suffix?

3. **How does the Python compiler know when to recompile a Python Standard Library module?**

   a. Module Name

   b. Timestamps

   c. File size

   d. Importing using: from math import sin

4. **Given a module called banking.py which has a function called deposit().**

Choose from the following, the code which would work:

a. from banking.py import deposit
   deposit(100)

b. import deposit from banking
   banking.deposit(200)

c. import banking
   banking.deposit(300)

d. from banking import deposit
   banking.deposit(400)

5. **Name the file which is created when a new Package is created?**
6. **True/False? Does this statement satisfy PEP8 requirements?**

   >>> import sys, math, banking

Solutions

Modules & Packages Quiz

1. Name the command used to load a Python module?

   Answer: import

2. Fill in the blank.

   Answer: Python modules have a .py suffix?

3. How does the Python compiler know when to recompile a Python Standard Library module?

   a. Module Name

   b. Timestamps

   c. File size

   d. Importing using: from math import sin

   Answer: b.Timestamps

Solutions

4. Given a module called banking.py which has a function called deposit().

Choose from the following, the code which would work:

a. from banking.py import deposit

deposit(100)

b. import deposit from banking

banking.deposit(200)

c. import banking

banking.deposit(300)

d. from banking import deposit

banking.deposit(400)

Answer: c

5. Name the file which is created when a new Package is created?

Answer: __init__.py

6. True/False? Does this statement satisfy PEP8 requirements?

import sys, math, banking

Answer: False (import on separate lines)

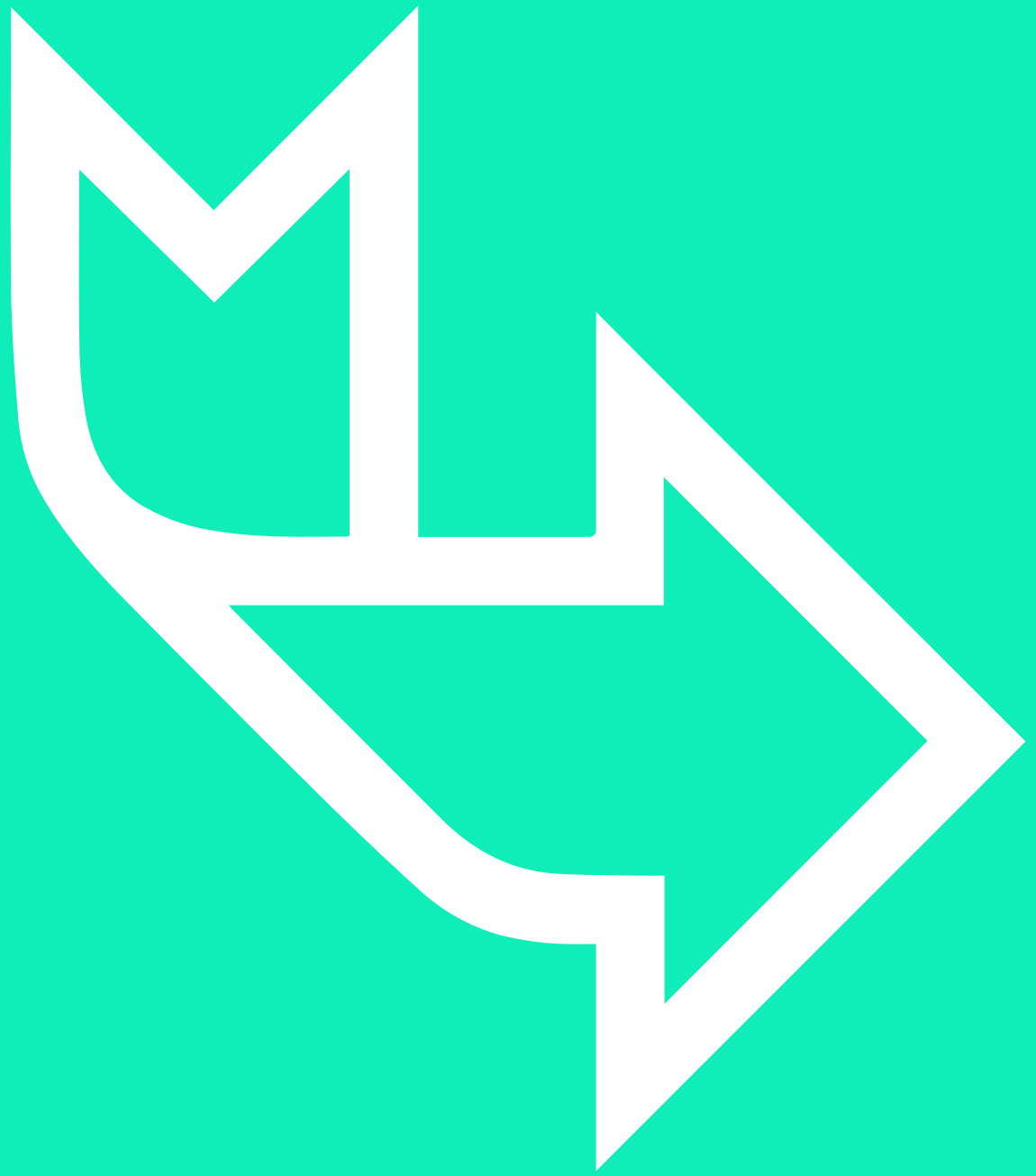
Labs

1. In this exercise, we are going to pull together some of the scripts and functions that you have already written and load them in as modules to a top-level menu program.

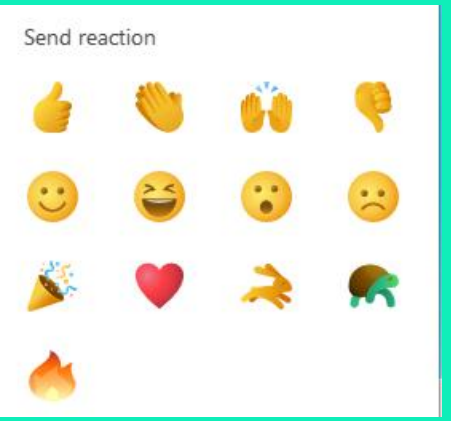
Create a new script **'C:\labs\menu.py'** that imports the appropriate functions from earlier lab exercises.

The script should display a menu like the following diagram, be contained within an infinite loop, and have a mechanism to quit out of the menu and the script. You should only import the functions that are required, which means you may have to modify the modules so that they have appropriate separate functions (and not solely a main() function).

Stretch Exercise: 2



END OF SECTION



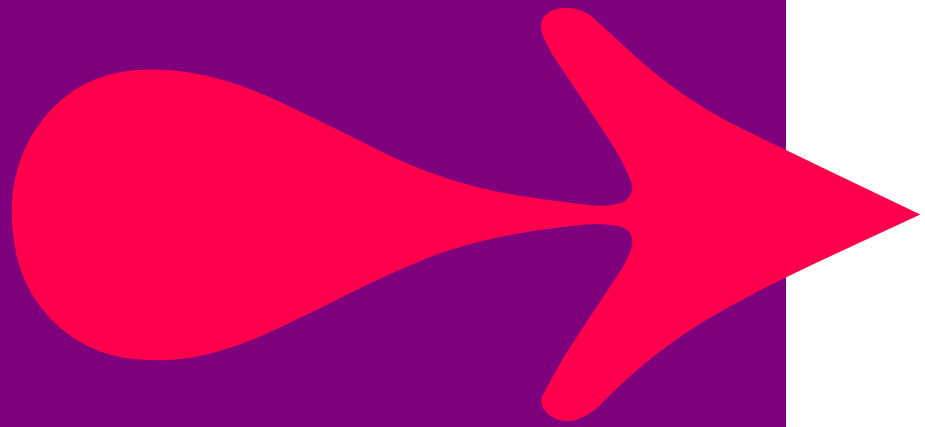
Taking code re-use up a level

- Modularity
- Building a Package
- Is it a program or a module?
- Namespaces
- The __main__ namespace trick
- Unit Testing

- To be able to break down a complex problem into more manageable sub-tasks.
- To create code that can be re-used, that is simple, and easily maintained.

Extension materials

Profiling



Profiling: Performance and Quality

- demo_profiling_before.py
- demo_profiling_after.py

The Python Standard Library module **'cProfile'** can be used to understand the way your code behaves and is especially useful with large and complex programs. In its simplest form, the output generated will tell you how many calls each of your functions and methods received, and how long was spent executing each function (by line number and file/module name).

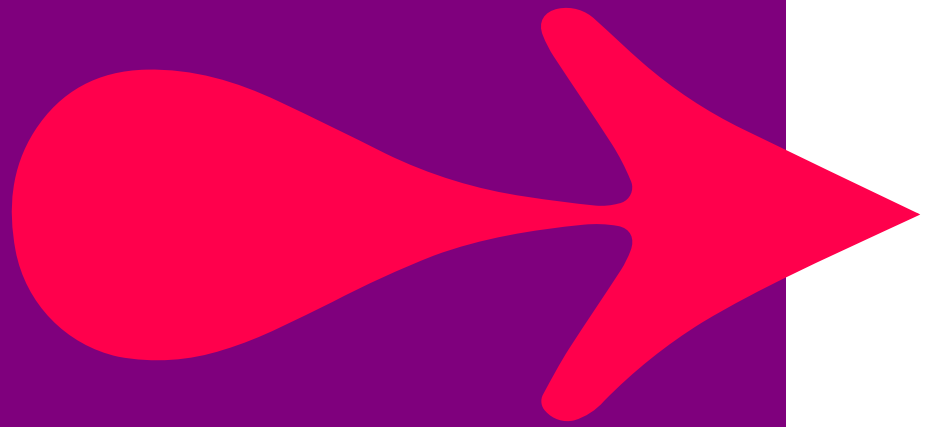
The analysis can be displayed to screen or written to a statistics file using a second parameter for the cProfile.run() function.

To analyse the statistics file:

```
C:\labs> python -m pstats stats.prof
% stats
```

Extension materials

Doc testing



Doc Testing: Automated Doc testing

- demo_doctesting.py

The **doctest** module searches for text that looks like interactive Python sessions in the Docstrings, and then executes those sessions to verify that they work as shown. This is useful for checking that a module's Docstrings are up to date, for regression testing when code changes and to embed a tutorial within the documentation.

Test in interactive Python session.

```
>>> add(4, 3)
7
```

Copy and paste into function Docstring.

REMINDER: TAKE A BREAK!

10.30 - 10.40
11.40 – 11.50
12.50 – 13.30
14.30 – 14.40
15.40 – 15.50