# Program Organisation: Namespaces, Modules, Imports

Intermediate Application Development

Otago Polytechnic
Dunedin, New Zealand
Kaiako: Tom Clark

# Introduction

My name is Tom. In many parts of the world that is a very common name. So when we talk about "Tom", who exactly do we mean?

- In this room, I'm probably the only person named Tom.
- I appear to be the only Tom Clark at OP, so on campus my "fully qualified" name identifies me.
- There are a lot of Tom Clarks in NZ, but if we say, "The Tom Clark at Otago Polytech," we're talking about me.

In the context of a running program, we have a similar problem.

# File: example.py

```
import otagopolytech

tom = "A really cool guy"
def enclose():
    tom = "I guess he's ok"
    def local():
        tom = "What a jerk"
        print(tom)
    print(tom)
    return local
opinion_of_tom = enclose()
opinion_of_tom()
print(tom)
```

What gets printed?

# Namespaces

In our example, the name `tom` gets used over and over, but they don't conflict because each one exists in a distinct *namespace*. In a running Python program several namespaces may exist at any one time.

When we use a name like `tom`, Python applies a set of rules for searching namespaces for that name.

# Global Namespace

```
import otagopolytech

tom = "A really cool guy"
def enclose():

...
```

The first occurrence of `tom` is in a *global* namespace. This name is meaningful anywhere in the file. The imported module `otagopolytechnic` actually defines a seperate, distinct global namespace.

# Local Namespace

```
...
def local():
    tom = "What a jerk"
    print(tom)
...
```

The innermost occurrence of `tom` is in a *local* namespace. This version of `tom` is only meaningful in the context of executing this function.

# Enclosing Namespace

```
    ...
def enclose():
    tom = "I guess he's ok"
    def local():
        tom = "What a jerk"
        print(tom)
    print(tom)
    return local
    ...
```

The function `enclose()` defines an *enclosing* namespace. This version of `tom` is meaningful in the context of the function `enclose()` **and** within the enclosed function `local()`.

# Builtin Namespace

```
import otagopolytech

tom = "A really cool guy"
def enclose():
    tom = "I guess he's ok"
    def local():
        tom = "What a jerk"
        print(tom)
   print(tom)
    return local
opinion_of_tom = enclose()
opinion_of_tom()
print(tom)
```

We didn't do anything to define `print()`. It is in the *builtin* namespace.

# LEGB

When the Python interpreter looks for a name like `tom`, it searches the namespaces in the order

1. Local
2. Enclosing
3. Global
4. Builtin

# Programming Activity

1. Pull the course materials repo.
2. Create a new branch, `06-practical` in your practicals repo.
3. Add a subdirectory, `06-practical` and copy `06-practical.ipynb` from the class materials into it.
4. Open a shell, cd to this directory, and run `jupyter notebook` to open the notebook. Complete the first questions.
5. We will discuss results in 30ish minutes.

# Imports

Suppose I have two files

```
mod.py
------------
num = 42
def foo():
    return 'bar'

main.py
------------
import mod

print(mod.num)
baz = mod.foo()
```

`mod.py` defines a *module*. The `import` brings the name `mod` into `main.py`'s global namespace and we can access its attributes there.

# Imports

We can also do this

```
mod.py
------------
num = 42
def foo():
    return 'bar'

main.py
------------
from mod import num

print(num)
```

In this case we just bring the name `num` into `main.py`'s namespace.

# Where does import find modules/packages?

When we use

```
import mod
```

The interpreter needs to find the module or package named `mod`. It searches the following locations in order.

1. `sys.modules` - a cache of loaded modules.
2. The current working directory from which the program was invoked.
3. Any directories listed in the `PYTHONPATH` environment variable.
4. A list of installation-dependent directories. You can see these by inspecting `sys.path`.

N.B.: These lists of locations can be modified at runtime, which is sometimes useful but also a security vulnerability if you are running untrusted code.

# Modules

As we've seen before, a Python *module* is just a file with Python code in it. It has it's own global namespace and anything defined at the global level may be imported.

```
mod.py
------------
num = 42
def foo():
    msg = 'bar'
    return msg
```

So in this case we can `import mod` and get access to `mod.num` and `mod.foo()`. We can't import `foo()`'s local variable, `msg`.

# Packages

We can bundle multiple modules together in a directory, and we call this a *package*

```
main.py
mypackage/
   |
   | mod1.py
   | mod2.py
```

Then, in `main.py` we can use

```
import mypackage.mod1
import mypackage.mod2
```

# Packages

If `mypackage/` contains a file called `__init__.py`, anything in that file can be imported with just `import mypackage`.

```
main.py
mypackage/
  | __init__.py
  | mod1.py
  | mod2.py
```

In `main.py`:

```
import mypackage
import mypackage.mod1
import mypackage.mod2
```

Notice that we still have to explicitly import `mypackage.mod1` and `mypackage.mod2`.

# Further Reading

- https://realpython.com/python-namespaces-scope/
- https://realpython.com/python-import/
- https://realpython.com/python-modules-packages/