6. How Modules and Functions Work

Topics:
- Modules and Functions
- More on Importing
- Call Frames

Let's Talk About Modules

What Are They?

M1.py

A module is a .py file that contains Python code.

The name of the module is the name of the file. This is the module M1.py.

We draw a module as a folder with a black outline.

Inside a Module

Kepler.py

A module may contain a single script.

A script will be shown as a rectangle with a red border.

Inside a Module

SimpleMath.py

A module may contain one or more function definitions.

Functions will be shown as rectangles with green borders.

Inside a Module

SimpleGraphics.py

A module may contain one or more data items.

These are referred to as global variables. They should be treated as constants whose values are never changed.

Data items will be shown as rectangles with blue borders.
A module may contain one or more data items and one or more functions.

Inside a Module

SimpleGraphics.py

- RED = [1.0, 0.0, 0.0]
- CYAN = [0.0, 1.0, 1.0]
- MakeWindow
- DrawRect

M.py

A module may contain one or more data items and one or more functions and a script.

But in this case, the script MUST be prefaced by

```python
if __name__ == '__main__':
```

Let's Talk About import

import M2:

M1.py

It means that code inside M1.py can reference the data and functions inside M2.py

M2.py

GlobVar1=3.14
GlobVar2=[1.0, 0.0]
func1(x)
func2(x, y)

What Does import Allow?

M1.py

import M2

GlobVar1=3.14
GlobVar2=[1.0, 0.0]
func1(x)
func2(x, y)

A function in M1.py could have a line like

```python
a = M2.func2(x, M2.GlobVar1)
```

What Does import Allow?

M1.py

import M2

GlobVar1=3.14
GlobVar2=[1.0, 0.0]
func1(x)
func2(x, y)

The script in M1.py could have a line like

```python
a = M2.func1(M2.GlobVar1)
```
One Way to Think About this...

```python
import M2

M1.py

GlobVar1=3.14
func1(x)
func2(x,y)

GlobVar2=[1,0,0]

M2.py

is like this...
```

Module M1.py contains a folder called M2. Need the "dot notation" to extract what is in M2.

What Does `import*` Allow?

```python
from M2 import*

M1.py

A function in M1.py could have a line like `a = func1(x,GlobalVar2)` No dot notation

GlobVar1=3.14
func1(x)
func2(x,y)

GlobVar2=[1,0,0]

M2.py

A script in M1.py could have a line like `a = func2(x,GlobalVar2)` No dot notation
```

One way to Think about this...

```python
from M2 import*

M1.py

It is as if GlobVar1, GlobVar2, func1, and func2 were defined in M1.py

GlobVar1=3.14
func1(x)
func2(x,y)

M2.py

is like this...
```
"Specific" Importing

A script in M1.py could have a line like `a = func2(3,4)`

No dot notation

A script in M1.py could NOT have a line like `a = func1(4)`

One way to think about this...

is like this...

It is as if `func2` was defined in M1.py

Using Stuff Within a Module

The functions and global variables in M.py can be used throughout M.py without the dot notation

There are rules about when a module M2.py can be imported by a module M1.py
Does this Always Work?

Yes, if M2.py is a module that is part of the CS 1110 Python installation, e.g.,

```
import M2
```

math    numpy    urllib2    string
scipy    PIL      random    timeit.

Does this Always Work?

No UNLESS M1.py and M2.py are each in the "current working directory".

import M2

Comments on "Launching" a Python Computation

In what follows, this will be how we indicate what's in the "current working directory"

And this will mean we are in the command shell and in the "current working directory"

```
cwd
```

```
cwd > python M1.py
```

Result: the script in M1.py is executed.

```
cwd > python M1.py
```

Result: the script in M1.py is executed.

```
cwd > python M1.py
```

Result: Nothing happens because there is no script in M1.py to execute.
important distinction

Distinguish between calling a function

\[ y = \sqrt{3} \]

and defining a function

```python
def sqrt(x):
    L = x
    L = (L + x/L) / 2
    return L
```

A function isn’t executed when it is defined.
Think of defining a function as setting up a formula that is to be used later.

somewhat like plugging into a formula

For the simple kind of fruitful functions that we have been considering, there is a substitution process.

Exactly how does it work?
We Use This Example...

```python
def T(s):
    r""" Returns as int the number of minutes from 12:00 to the time specified by s."
    PreC: s is a length-5 string of the form 'hh:mm' that specifies the time.
    h = int(s[:2])
    m = int(s[3:])
    if h<12:
        z = 60*h+m
    else:
        z = m
    return z
```

A Script

```python
s1 = '11:15'
s2 = '12:05'
x = T(s1)
y = T(s2)

if y>=x:
    numMin = y-x
else:
    numMin = (y+720)-x
```

This assigns to `numMin` the number of minutes in a class that starts at the time specified by `s1` and ends at the time specified by `s2`.

Prints the number of minutes in a class that starts at the time specified by `s1` and ends at the time specified by `s2`. Let us step through its execution.

s1 = '11:15'
s2 = '12:05'
x = T(s1)
y = T(s2)

if y>=x:
    numMin = y-x
else:
    numMin = (y+720)-x

print numMin

Function call: The defined function `T` will now be asked to process the value in `s1`.
Let's track what happens...
Python code:

```python
def T(s):
    h = int(s[:2])
    m = int(s[3:])
    if h<12:
        z = 60*h+m
    else:
        z = m
    return z
```

Function call:

We open up a "call frame" that shows the "key players" associated with the function.

```
s1 = '11:15'
s2 = '12:05'
x = T(s1)
y = T(s2)
if y>=x:
    numMin = y - x
else:
    numMin = (y+720) - x
print numMin
```

Return is a special variable. Will house the value to return.
We step through the function body, business as usual.

The return value is shipped back the to red dot instruction.
s1 = '11:15'
s2 = '12:05'
x = T(s1)
y = T(s2)
if y >= x:
    numMin = y - x
else:
    numMin = (y + 720) - x
print numMin

def T(s):
    h = int(s[:2])
m = int(s[3:])
if h < 12:
    z = 60 * h + m
else:
    z = m
return z

The function call is over. The Call Frame "disappears"...

The value of the argument (housed in s2) is substituted

Execution of the function body starts.

We step through the function body.
s1 = '11:15'
s2 = '12:05'
x = T(s1)
y = T(s2)
if y>=x:
    numMin = y-x
else:
    numMin = (y+720)-x
print(numMin)

\[
\text{def } T(s): \\
\text{    } h = \text{int}(s[:2]) \\
\text{    } m = \text{int}(s[3:]) \\
\text{    } \text{if h<12:} \\
\text{        } z = 60*h+m \\
\text{    } \text{else:} \\
\text{        } z = m \\
\text{    } \text{return } z \\
\]

We step through the function body.

That value is sent back to the red dot.

Function call is over. Call Frame disappears. Red dot moves on

The script is over. Global space disappears.