6. How Functions Work and Are Accessed

Topics:

- Modules and Functions
- More on Importing
- Call Frames

Let's Talk About Modules

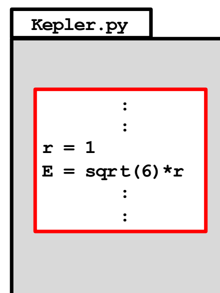
What Are They?

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.

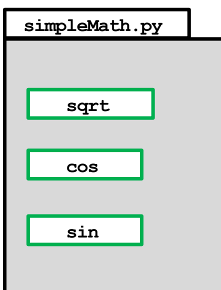
Inside a Module



A module may contain a single script.

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

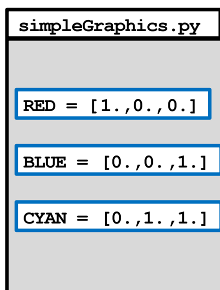
Inside a Module



A module may contain one or more function definitions.

Functions will be shown as rectangles with green borders.

Inside a Module

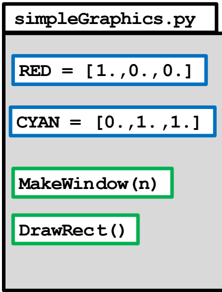


A module may contain one or more data items.

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

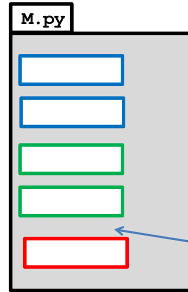
Data items will be shown as rectangles with blue borders.

Inside a Module



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

Inside a Module



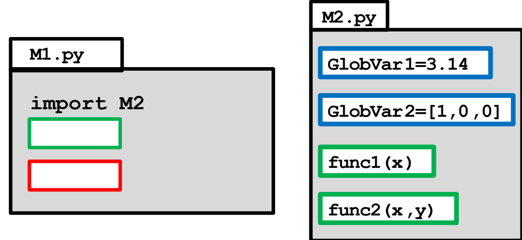
A module may contain one or more data items and one or more functions and a script.

But in this case, the script MUST be prefaced by

`if __name__ == __main__:`

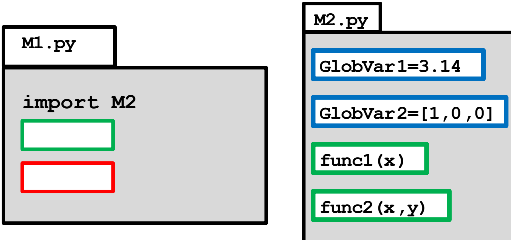
Let's Talk About `import`

What Does `import` Allow?



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

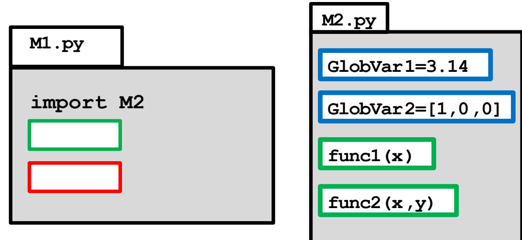
What Does `import` Allow?



A function [green box] in **M1.py** could have a line like

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

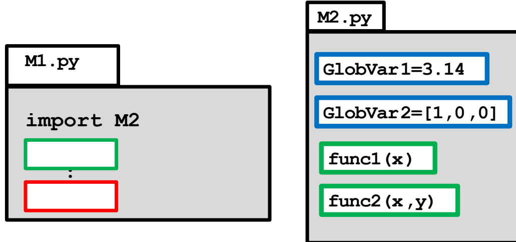
What Does `import` Allow?



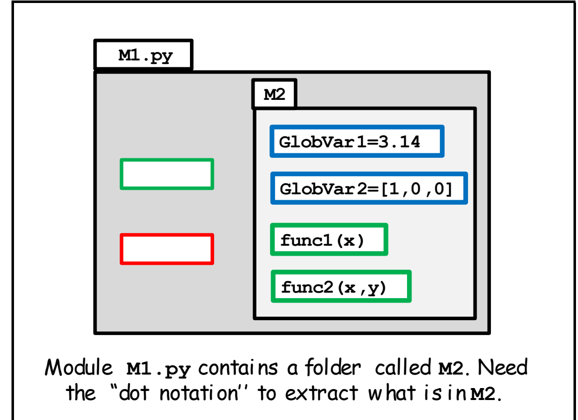
The script [red box] in **M1.py** could have a line like

`a = M2.func1(M2.GlobVar1)`

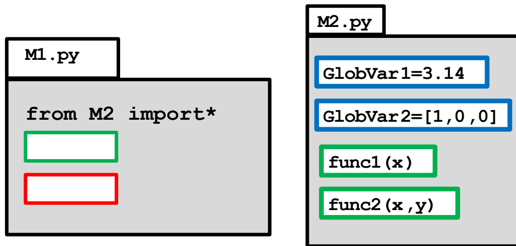
One Way to Think About this...



is like this...



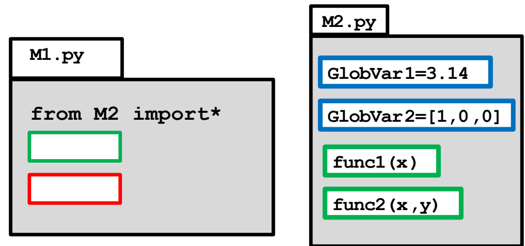
What Does import* Allow?



A function [green box] in `M1.py` could have a line like `a = func1(x,GlobalVar2)`

No dot notation

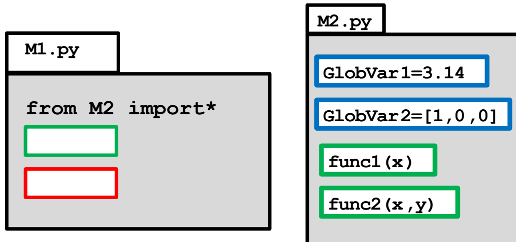
What Does import* Allow?



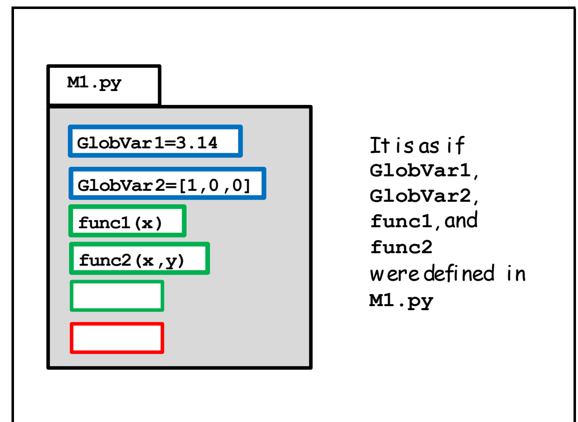
A script [red box] in `M1.py` could have a line like `a = func2(x,GlobalVar2)`

No dot notation

One way to Think about this...



is like this...



"Specific" Importing

M1.py

```
from M2 import func2
```

M2.py

```
GlobVar1=3.14
GlobVar2=[1,0,0]
func1(x)
func2(x,y)
```

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

No dot notation

"Specific" Importing

M1.py

```
from M2 import func2
```

M2.py

```
GlobVar1=3.14
GlobVar2=[1,0,0]
func1(x)
func2(x,y)
```

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

No dot notation

One way to think about this…

M1.py

```
from M2 import func2
```

M2.py

```
GlobVar1=3.14
GlobVar2=[1,0,0]
func1(x)
func2(x,y)
```

is like this…

M1.py

```
func2(x,y)
```

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

Using Stuff Within a Module

M.py

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?

M1.py

```
import M2
   :
```

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

```
math     numpy    urllib2   string
scipy    PIL      random    timeit.
```

Does this Always Work?

M1.py

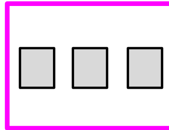
```
import M2
   :
```

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

.

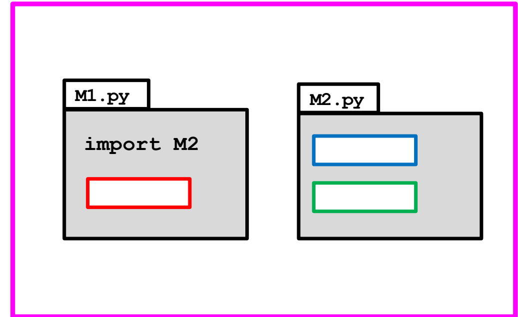
Comments on "Launching" a Python Computation

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



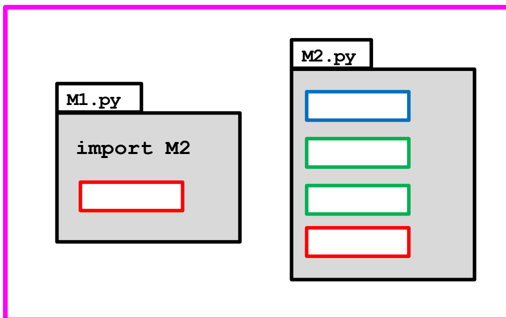
And this will mean we are in the command shell and in the "current working directory"

`cwd >`



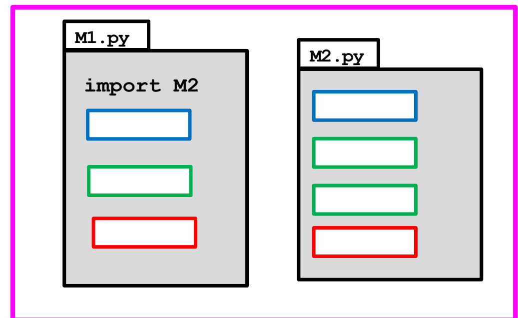
`cwd > python M1.py`

The script in `M1.py` is executed



`cwd > python M1.py`

The script in `M1.py` is executed



`cwd > python M1.py`

The script in `M1.py` is executed

M1.py

```
import M2
```

M2.py

```
if __name__ == __main__:
```

cwd > python M1.py

The script in M2.py is NOT executed

M1.py

```
import M2
```

M2.py

cwd > python M1.py

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

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

M1.py

```
import M2
```

M2.py

cwd > python M1.py

Error. Python cannot find M2

M1.py

```
import M2
```

M2.py

cwd > python M1.py

Error. Python cannot find M1

M1.py

```
import M2
```

M2.py

cwd > python M2.py

Fine. M2.py does not need M1.py

Now let's consider what happens when a function is called.

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

Exactly how does it work?

A Function

```
def T(s):
""" 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

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

function call

function call

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**.

A Script

```
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
```

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.

1

```
● 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
```

The red dot indicates the next thing to do in the script.

s1 →	
s2 →	
x →	
y →	
numMin →	

This box is called Global Space. It includes all the variables associated with the script.

2

```
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
```

s1 →	'11:15'
s2 →	
x →	
y →	
numMin →	

3

```
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
```

s1 → '11:15'
s2 → '12:05'
x →
y →
numMin →

Function call

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

---

3

```
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
```

s1 → '11:15'
s2 → '12:05'
x →
y →
numMin →

s →
h →
m →
z →
return →

The variable s is the function's parameter

3

```
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
```

s1 → '11:15'
s2 → '12:05'
x →
y →
numMin →

s →
h →
m →
z →
return →

The variables h, m, and z is the function's local variables

3

```
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
```

s1 → '11:15'
s2 → '12:05'
x →
y →
numMin →

s →
h →
m →
z →
return →

return is a special variable. Will house the value to return

4

```
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
```

s1 → '11:15'
s2 → '12:05'
x →
y →
numMin →

s →
h →
m →
z →
return →

Control passes from the red dot to the blue dot

5

```
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
```

s1 → '11:15'
s2 → '12:05'
x →
y →
numMin →

s → '11:15'
h →
m →
z →
return →

Assign the argument value (housed in s1) to the parameter s.

5

```
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
```

Variable	Value
s1	'11:15'
s2	'12:05'
x	
y	
numMin	

Variable	Value
s	'11:15'
h	
m	
z	
return	

Assign the argument value (housed in s1) to the parameter s.

6

```
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
```

Variable	Value
s1	'11:15'
s2	'12:05'
x	
y	
numMin	

Variable	Value
s	'11:15'
h	11
m	
z	
return	

We step through the function body. Business as usual.

7

```
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
```

Variable	Value
s1	'11:15'
s2	'12:05'
x	
y	
numMin	

Variable	Value
s	'11:15'
h	11
m	15
z	
return	

We step through the function body. Business as usual.

8

```
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
```

Variable	Value
s1	'11:15'
s2	'12:05'
x	
y	
numMin	

Variable	Value
s	'11:15'
h	11
m	15
z	675
return	

We step through the function body. Business as usual.

9

```
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
```

Variable	Value
s1	'11:15'
s2	'12:05'
x	675
y	
numMin	

Variable	Value
s	'11:15'
h	11
m	15
z	675
return	675

The return value is shipped back the to red dot instruction.

10

```
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
```

Variable	Value
s1	'11:15'
s2	'12:05'
x	675
y	
numMin	

Variable	Value
s	'11:15'
h	11
m	15
z	675
return	675

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

Slide 11

```
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
```

s1 →	'11:15'
s2 →	'12:05'
x →	675
y →	
numMin →	

Another function Call!

And the red dot moves to the next statement in the script

Slide 12

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

s1 →	'11:15'
s2 →	'12:05'
x →	675
y →	
numMin →	

s →	
h →	
m →	
z →	
return →	

We open up the Call Frame

Slide 13

s1 →	'11:15'
s2 →	'12:05'
x →	675
y →	
numMin →	

s →	'12:05'
h →	
m →	
z →	
return →	

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

Slide 13

s →	'12:05'
h →	
m →	
z →	
return →	

Execution of the function body starts.

Slide 14

s1 →	'11:15'
s2 →	'12:05'
x →	675
y →	
numMin →	

s →	'12:05'
h →	12
m →	
z →	
return →	

We step through the function body

Slide 15

s1 →	'11:15'
s2 →	'12:05'
x →	675
y →	
numMin →	

s →	'12:05'
h →	12
m →	5
z →	
return →	

We step through the function body.

16

```
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
```

s1	'11:15'
s2	'12:05'
x	675
y	
numMin	

s	'12:05'
h	12
m	5
z	5
return	

We step through the function body.

17

```
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
```

s1	'11:15'
s2	'12:05'
x	675
y	
numMin	

s	'12:05'
h	12
m	5
z	5
return	5

The value in z is to be returned

18

```
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
```

s1	'11:15'
s2	'12:05'
x	675
y	5
numMin	

s	'12:05'
h	12
m	5
z	5
return	5

That value is sent back to the red dot.

19

```
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
```

s1	'11:15'
s2	'12:05'
x	675
y	5
numMin	50

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

20

```
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
```

s1	'11:15'
s2	'12:05'
x	675
y	5
numMin	50

50

The script is over. Global space disappears.

21

50

The script is over. Global space disappears.

Key Points

1. Flow of Control

When a function is called, control is passed to the function. The caller waits for the results to be returned before proceeding. (The red dot holds up until the blue dot is finished.)

2. Substitution

At the time of the call, each input parameter takes on the value of the corresponding the argument.