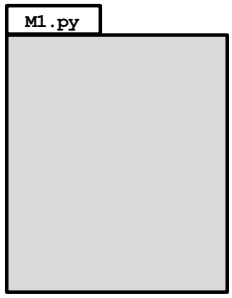
6. How Modules and Functions Work

Topics:

- Modules and Functions
- More on Importing
- Call Frames

Let's Talk About Modules

What Are They?

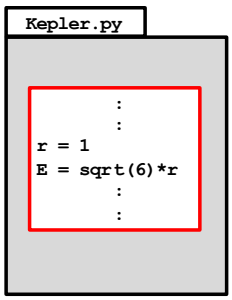


A module is a .py file that contains Python code

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.

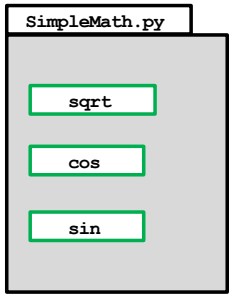
Inside a Module



A module may contain a single script.

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

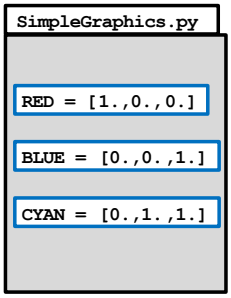
Inside a Module



A module may contain one or more function definitions.

Functions will be shown as rectangles with green borders.

Inside a Module

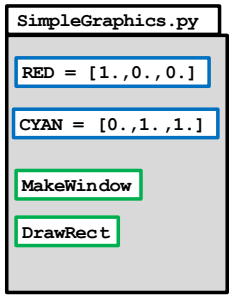


A module may contain one or more data items.

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

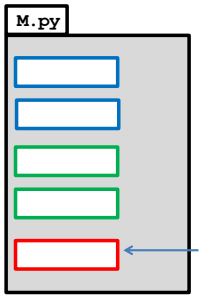
Data items will be shown as rectangles with blue borders.

Inside a Module



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

Inside a Module



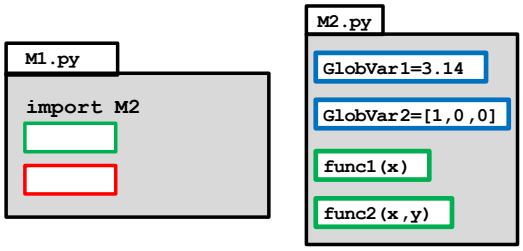
A module may contain one or more data items and one or more functions and a script.

But in this case, the script MUST be prefaced by

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

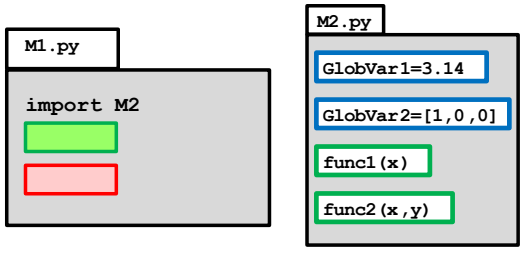
Let's Talk About `import`

What Does `import` Allow?



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

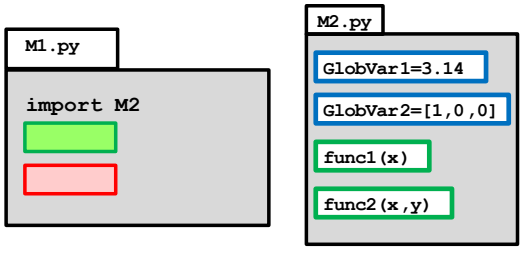
What Does `import` Allow?



A function in `M1.py` could have a line like

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

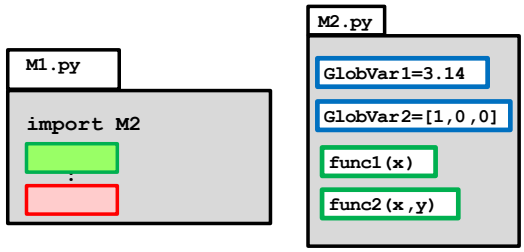
What Does `import` Allow?



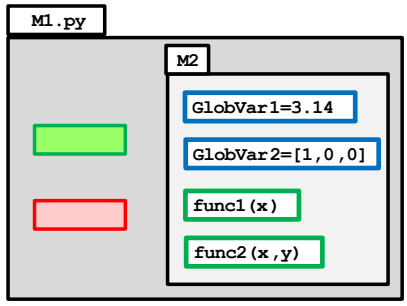
The script in `M1.py` could have a line like

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

One Way to Think About this…

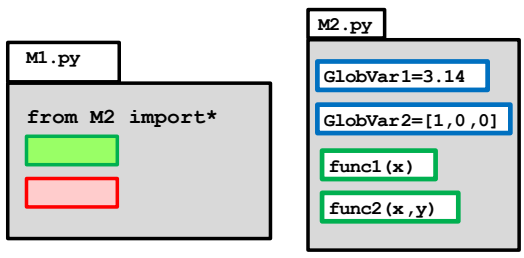


is like this…



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

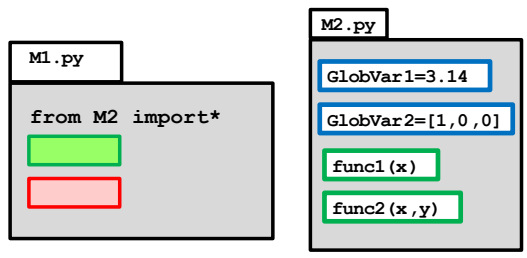
What Does `import*` Allow?



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

No dot notation

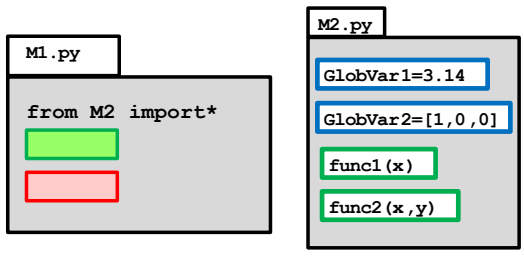
What Does `import*` Allow?



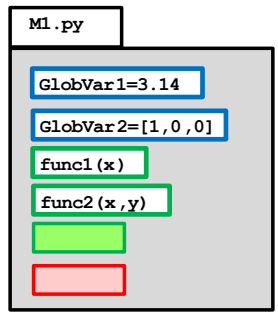
A script in `M1.py` could have a line like `a = func2(x,GlobalVar2)`

No dot notation

One way to Think about this…

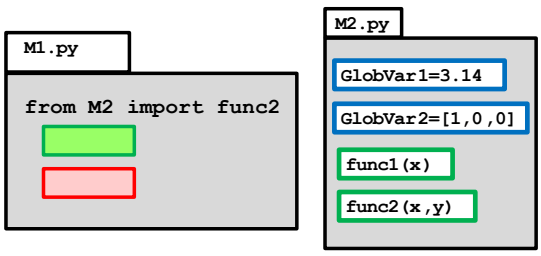


is like this…



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

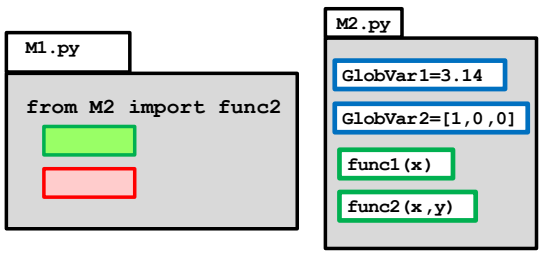
"Specific" Importing



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

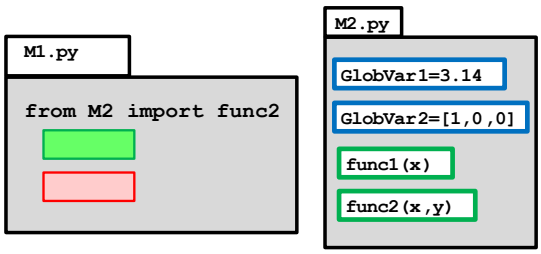
No dot notation

"Specific" Importing

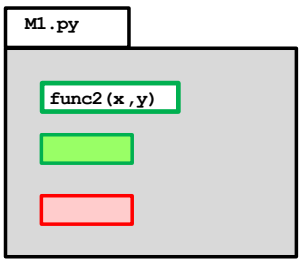


A script ▭ in `M1.py` could NOT have a line like `a = func1(4)`

One way to think about this…

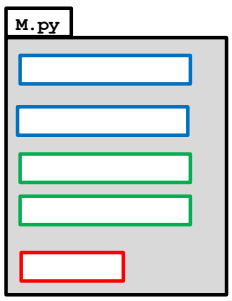


is like this…



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?

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

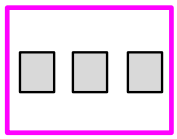
```
import M2
   :
```

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

.

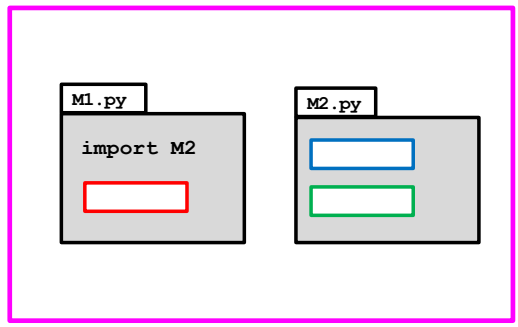
Comments on "Launching" a Python Computation

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

➡ 

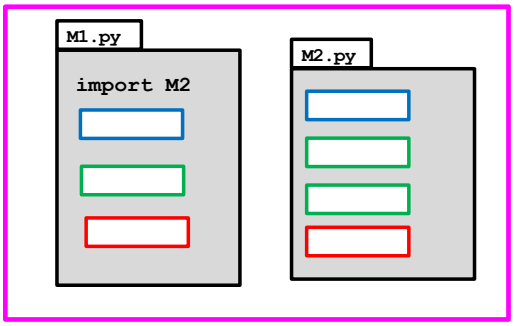
And this will mean we are in the command shell and in the "current working directory"

➡ `cwd >`



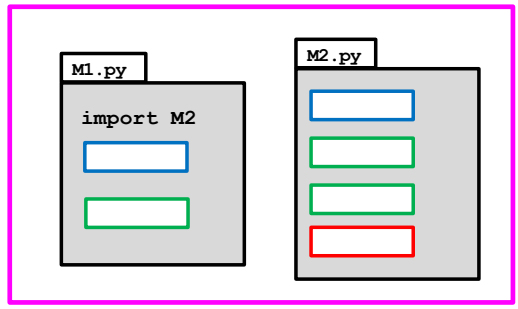
`cwd > python M1.py`

Result: the script in `M1.py` is executed.



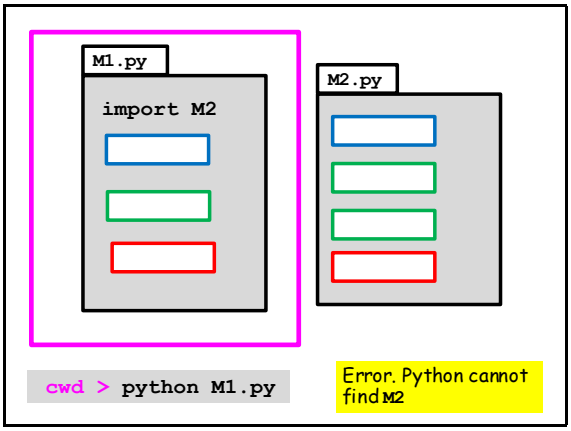
`cwd > python M1.py`

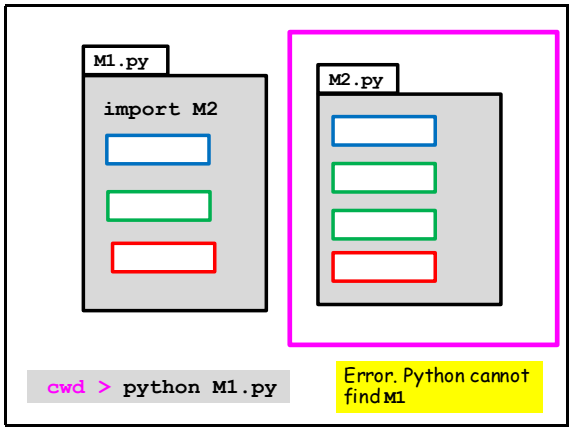
Result: the script in `M1.py` is executed. The script in `M2.py` is not executed

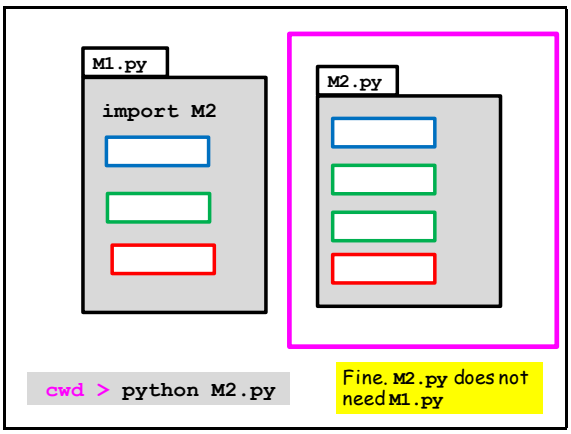


`cwd > python M1.py`

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

```
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.

We now focus our attention on the mechanics behind function calls.

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…

```
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

function call
function call

```
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**.

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
```

s1 ⟶
s2 ⟶
x ⟶
y ⟶
numMin ⟶

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

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
```

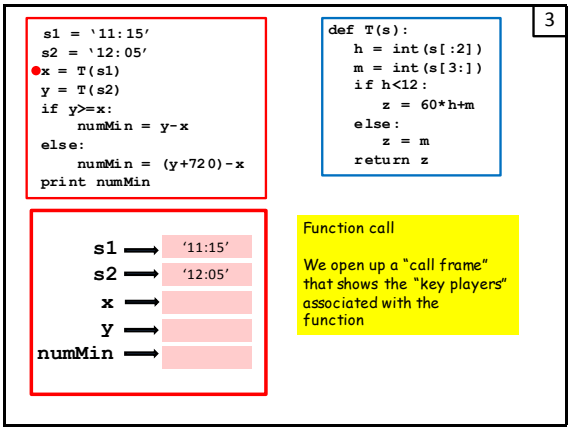
```
def T(s):
   h = int(s[:2])
   m = int(s[3:])
   if h<12:
      z = 60*h+m
   else:
      z = m
   return z
```

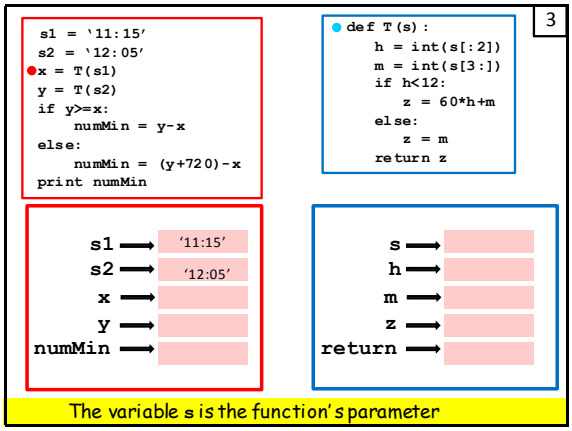
s1 ⟶ '11:15'
s2 ⟶ '12:05'
x ⟶
y ⟶
numMin ⟶

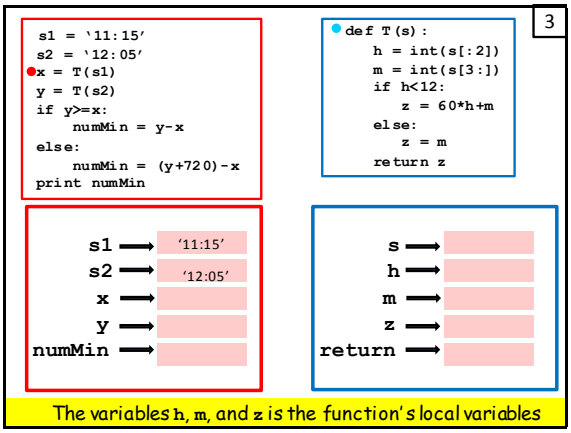
Function call

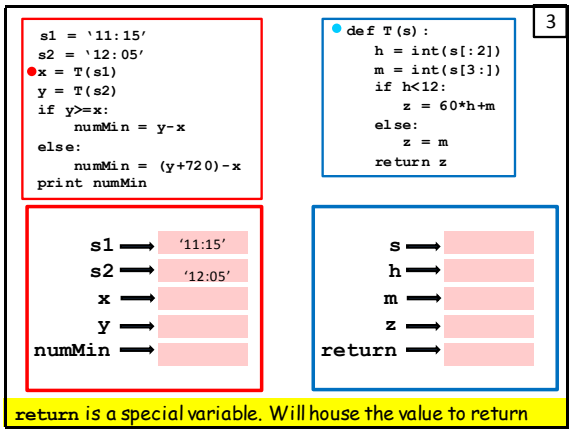
The defined function T will now be asked to process the value in s1.

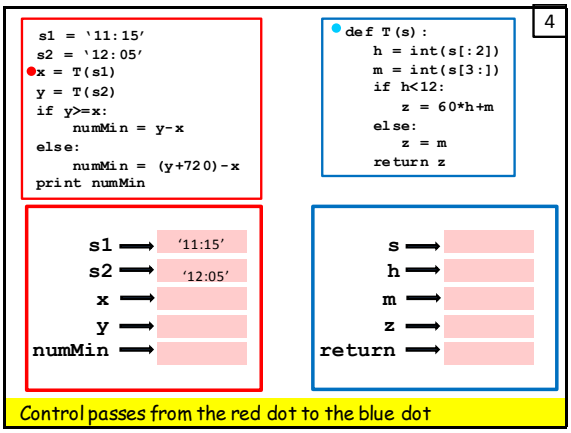
Let's track what happens…

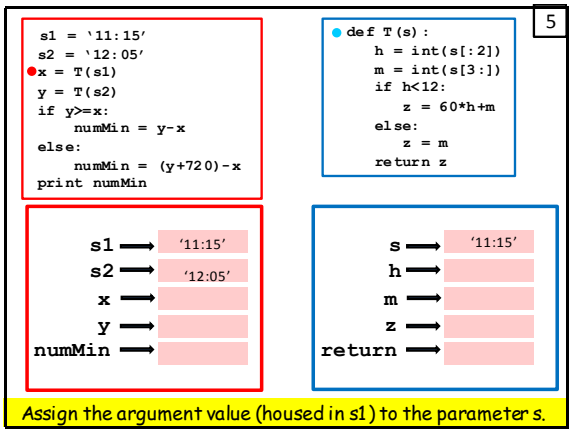

3
s1 = '11:15'
s2 = '12:05'
x = T(s1)
y = T(s2)
if y>=x:
numMin = y-x
else:
numMin = (y+720)-x
print numMin
def T(s):
h = int(s[:2])
m = int(s[3:])
if h<12:
z = 60*h+m
else:
z = m
return z
s1 '11:15'
s2 '12:05'
x
y
numMin
Function call
We open up a "call frame" that shows the "key players" associated with the function

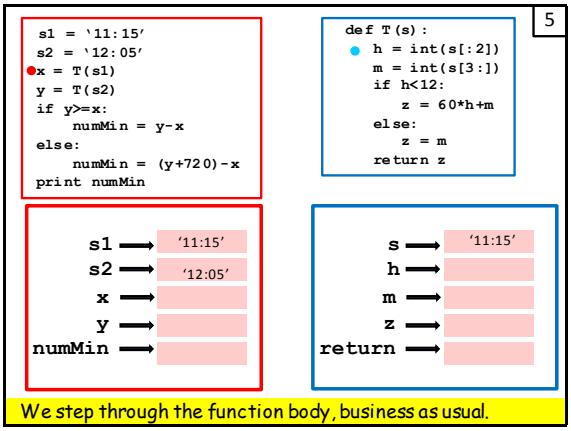

3
s1 = '11:15'
s2 = '12:05'
x = T(s1)
y = T(s2)
if y>=x:
numMin = y-x
else:
numMin = (y+720)-x
print numMin
def T(s):
h = int(s[:2])
m = int(s[3:])
if h<12:
z = 60*h+m
else:
z = m
return z
s1 '11:15'
s2 '12:05'
x
y
numMin
s
h
m
z
return
The variable s is the function's parameter

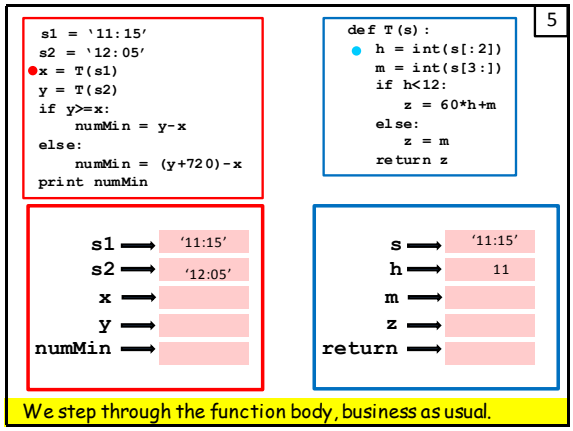

3
s1 = '11:15'
s2 = '12:05'
x = T(s1)
y = T(s2)
if y>=x:
numMin = y-x
else:
numMin = (y+720)-x
print numMin
def T(s):
h = int(s[:2])
m = int(s[3:])
if h<12:
z = 60*h+m
else:
z = m
return z
s1 '11:15'
s2 '12:05'
x
y
numMin
s
h
m
z
return
The variables h, m, and z is the function's local variables

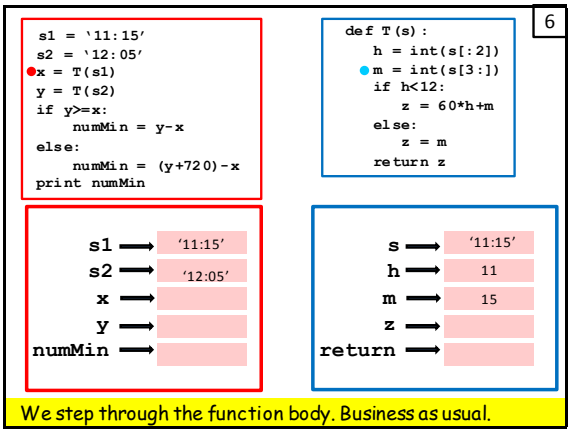

3
s1 = '11:15'
s2 = '12:05'
x = T(s1)
y = T(s2)
if y>=x:
numMin = y-x
else:
numMin = (y+720)-x
print numMin
def T(s):
h = int(s[:2])
m = int(s[3:])
if h<12:
z = 60*h+m
else:
z = m
return z
s1 '11:15'
s2 '12:05'
x
y
numMin
s
h
m
z
return
return is a special variable. Will house the value to return

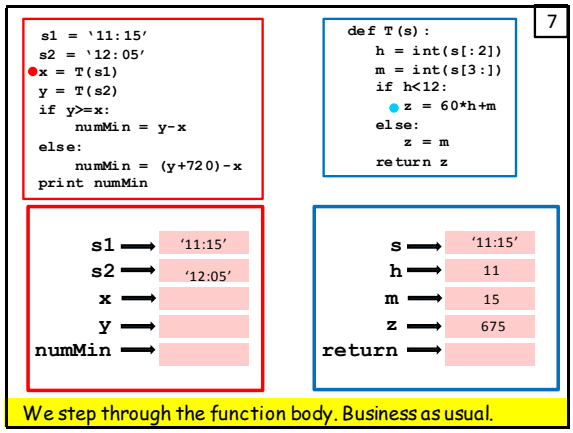

4
s1 = '11:15'
s2 = '12:05'
x = T(s1)
y = T(s2)
if y>=x:
numMin = y-x
else:
numMin = (y+720)-x
print numMin
def T(s):
h = int(s[:2])
m = int(s[3:])
if h<12:
z = 60*h+m
else:
z = m
return z
s1 '11:15'
s2 '12:05'
x
y
numMin
s
h
m
z
return
Control passes from the red dot to the blue dot

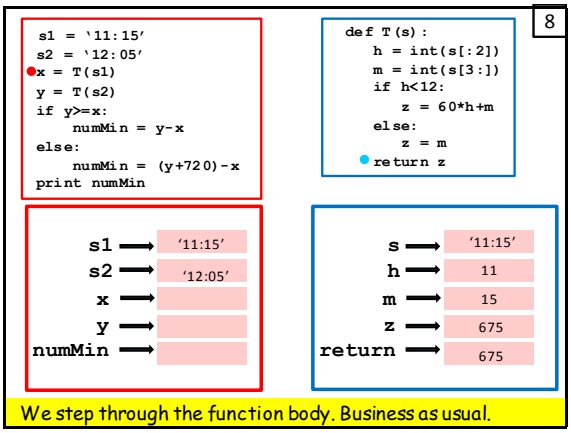

5
s1 = '11:15'
s2 = '12:05'
x = T(s1)
y = T(s2)
if y>=x:
numMin = y-x
else:
numMin = (y+720)-x
print numMin
def T(s):
h = int(s[:2])
m = int(s[3:])
if h<12:
z = 60*h+m
else:
z = m
return z
s1 '11:15'
s2 '12:05'
x
y
numMin
s '11:15'
h
m
z
return
Assign the argument value (housed in s1) to the parameter s.

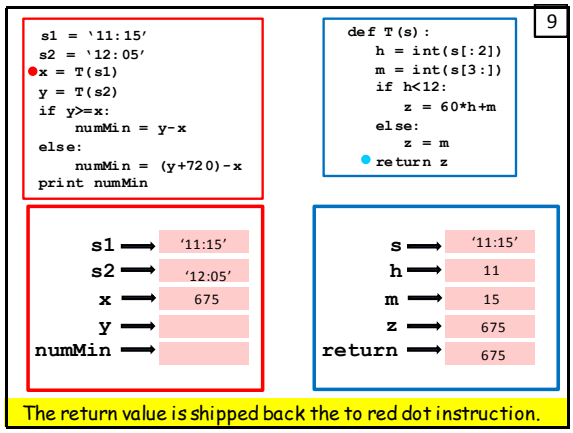

5
s1 = '11:15'
s2 = '12:05'
x = T(s1)
y = T(s2)
if y>=x:
numMin = y-x
else:
numMin = (y+720)-x
print numMin
def T(s):
h = int(s[:2])
m = int(s[3:])
if h<12:
z = 60*h+m
else:
z = m
return z
s1 '11:15'
s2 '12:05'
x
y
numMin
s '11:15'
h
m
z
return
We step through the function body, business as usual.


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 11
m
z
return
We step through the function body, business as usual.


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
s1 '11:15'
s2 '12:05'
x
y
numMin
s '11:15'
h 11
m 15
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
s1 '11:15'
s2 '12:05'
x
y
numMin
s '11:15'
h 11
m 15
z 675
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
s1 '11:15'
s2 '12:05'
x
y
numMin
s '11:15'
h 11
m 15
z 675
return 675
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
s1 '11:15'
s2 '12:05'
x 675
y
numMin
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
```

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

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

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

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
```

Another function Call!

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

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

12

```
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	
h	
m	
z	
return	

We open up the Call Frame

13

```
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	
m	
z	
return	

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

13

```
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	
z	
return	

Execution of the function body starts.

14

```
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	
return	

We step through the function body

15

```
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.

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	5

We step through the function body.

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.