

# 6. How Modules and Functions Work

Topics:

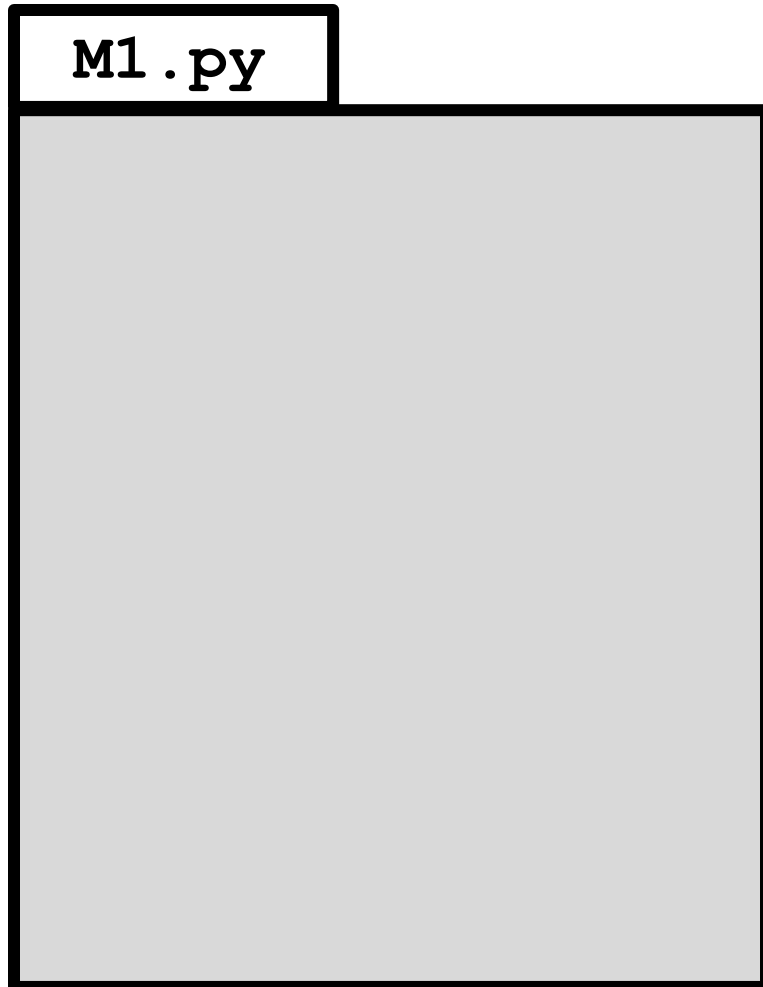
Modules and Functions

More on Importing

Call Frames

Let's Talk About Modules

# What Are They?

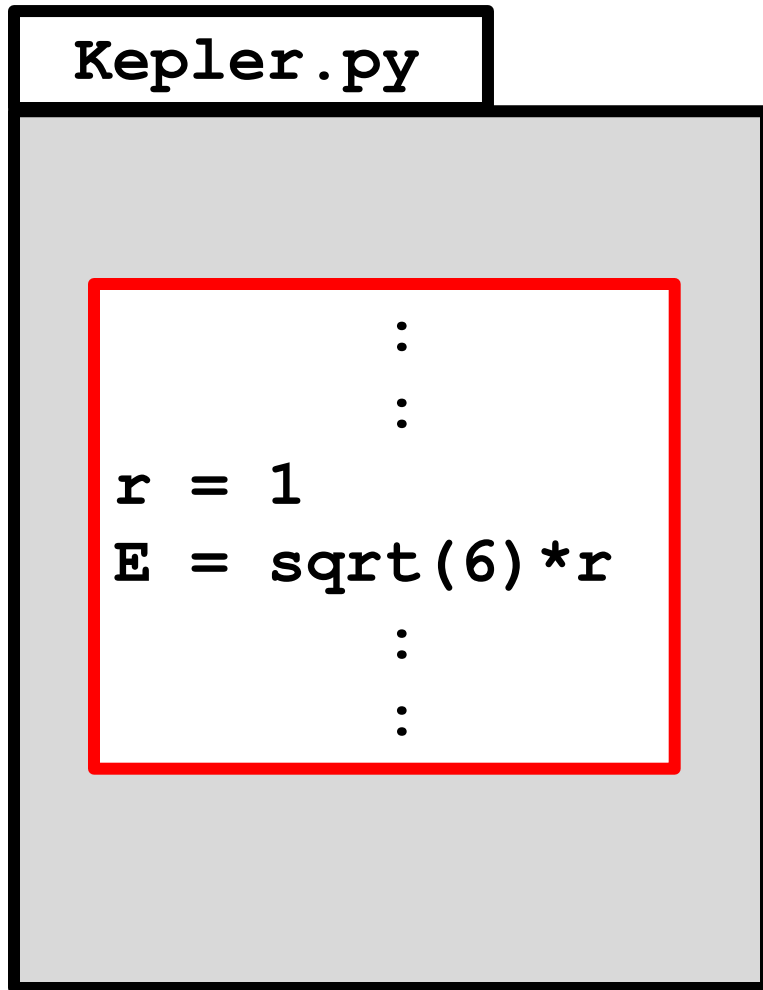


A module is a .py file that contains Python code

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



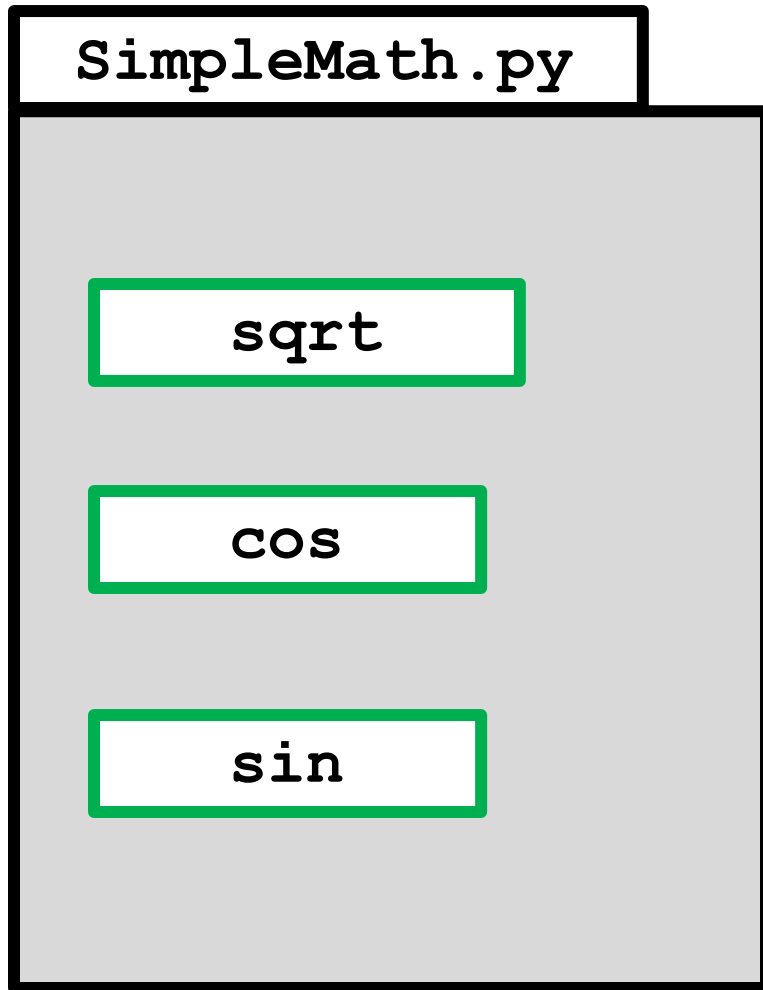
The diagram shows a gray rectangular box representing a Python module. At the top left of the box is a white tab labeled "Kepler.py". Inside the gray box, there is a white rectangular area with a red border, representing a script. The script contains the following code:

```
        :  
        :  
r = 1  
E = sqrt(6) * r  
        :  
        :
```

A module may contain a single script.

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

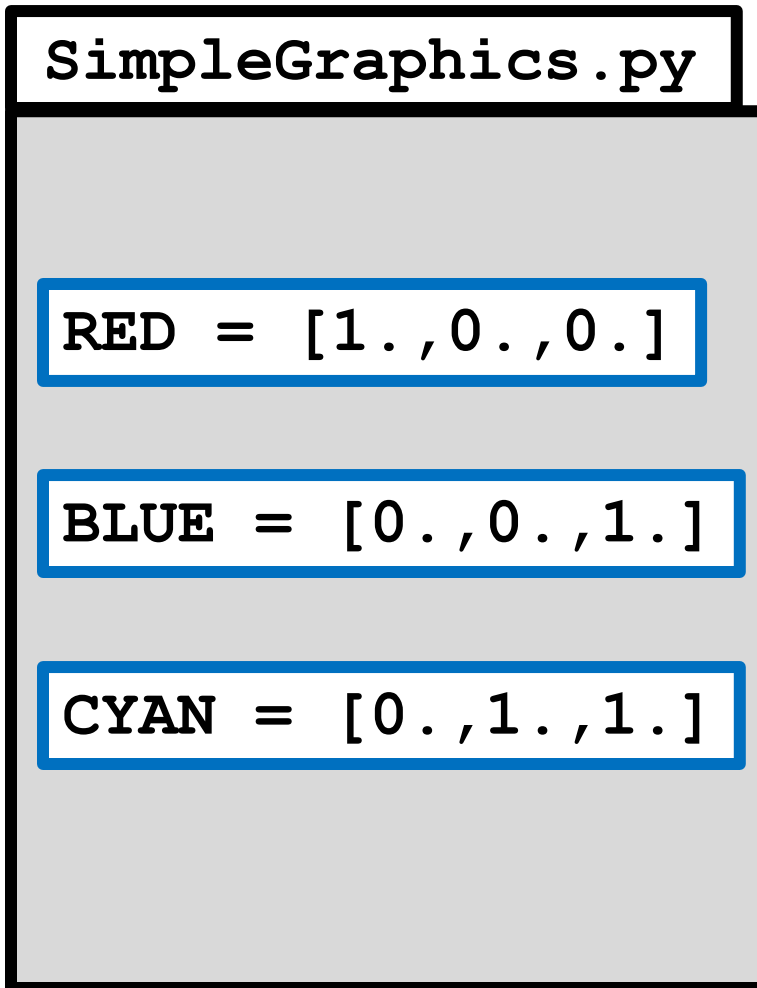
# Inside a Module



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

```
SimpleGraphics.py
```

```
RED = [1.,0.,0.]
```

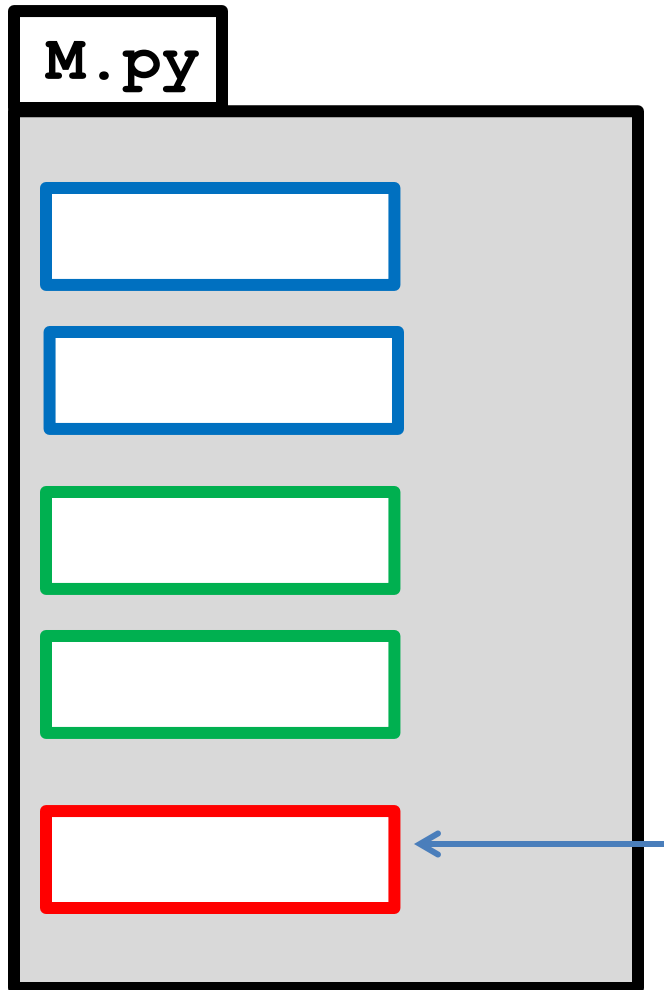
```
CYAN = [0.,1.,1.]
```

```
MakeWindow
```

```
DrawRect
```

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.

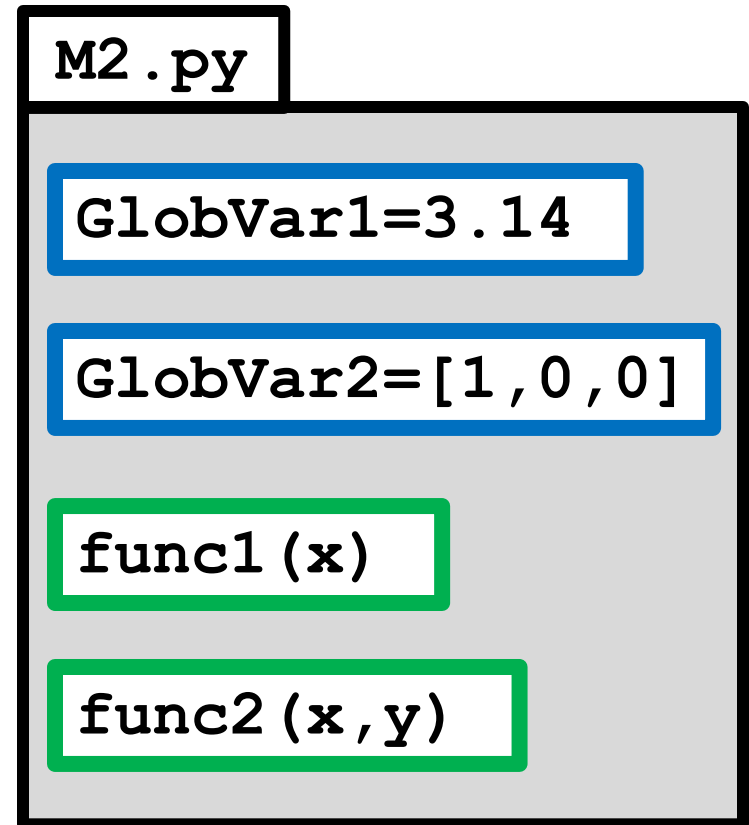
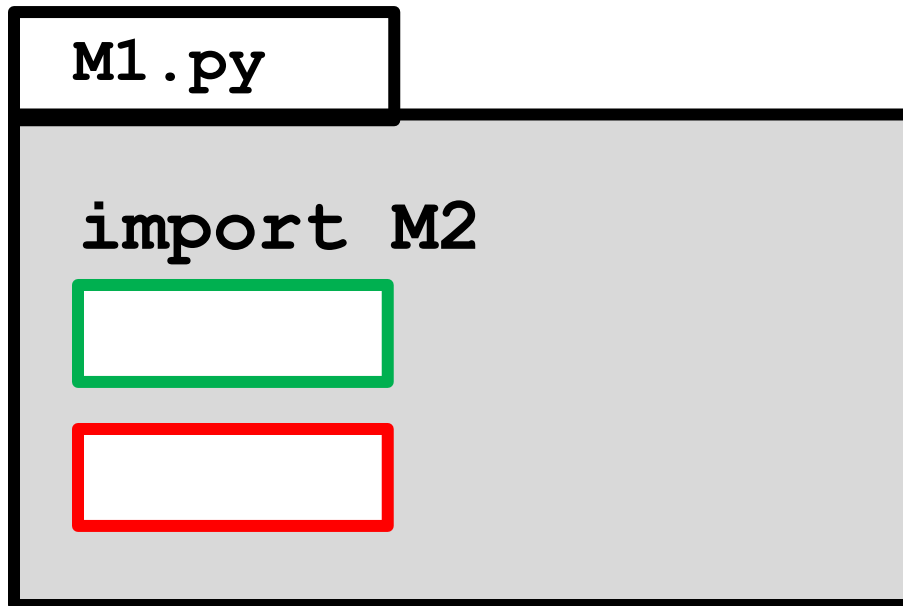
But in this case, the script  
**MUST** be prefaced by

```
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

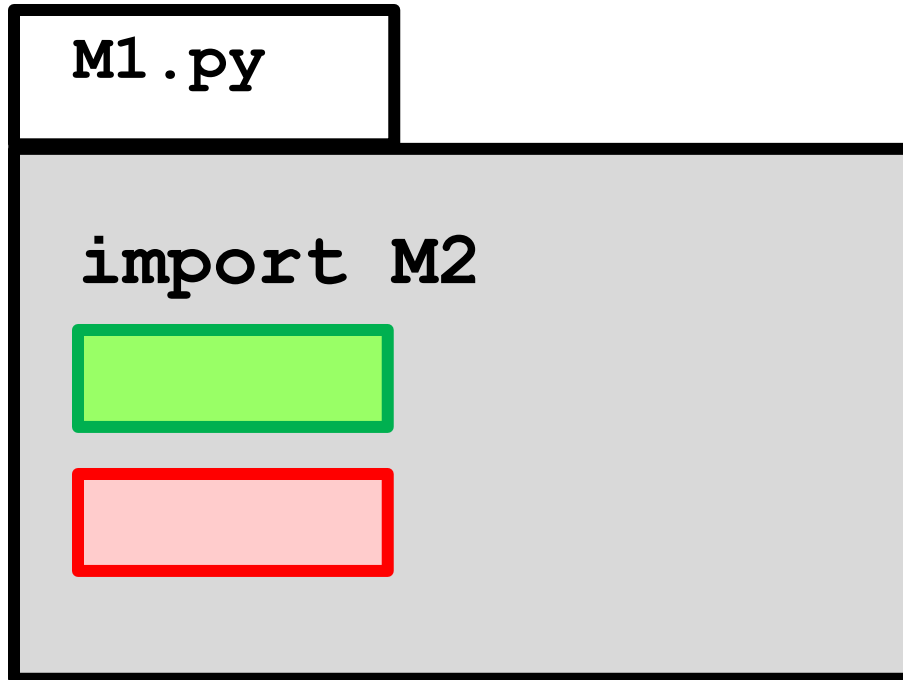
# What Does `import` Allow?

```
M1.py
```

```
import M2
```

```
[ ]
```

```
[ ]
```

A diagram representing a Python file named M1.py. It has a title box at the top containing the text 'M1.py'. Below the title is a large grey rectangular area representing the file's content. The first line of code is 'import M2'. Below this line are two rectangular boxes: a light green one and a light red one, both with black outlines, representing placeholders for code.

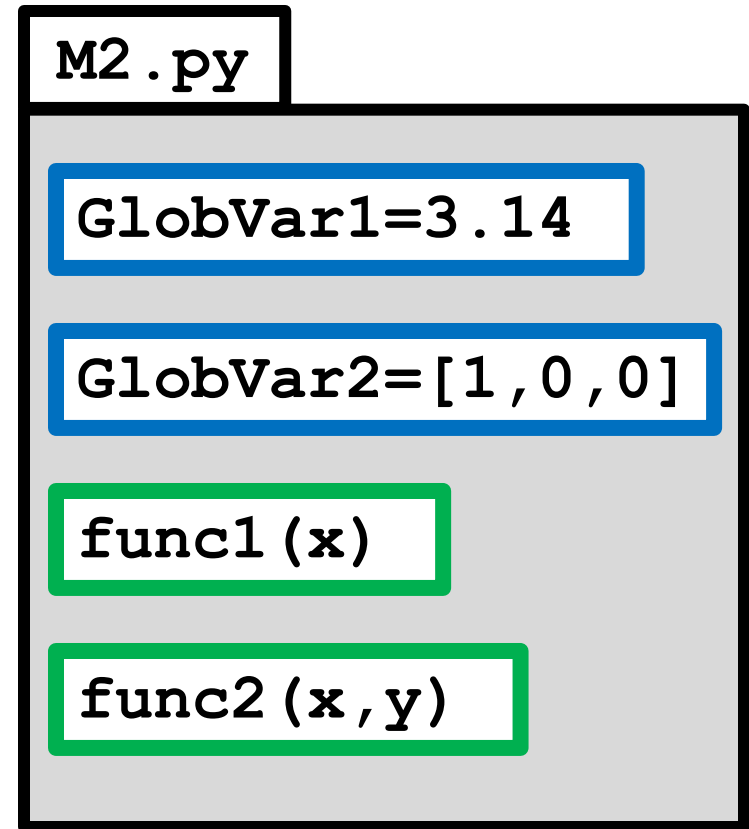
```
M2.py
```

```
GlobVar1=3.14
```

```
GlobVar2=[1,0,0]
```

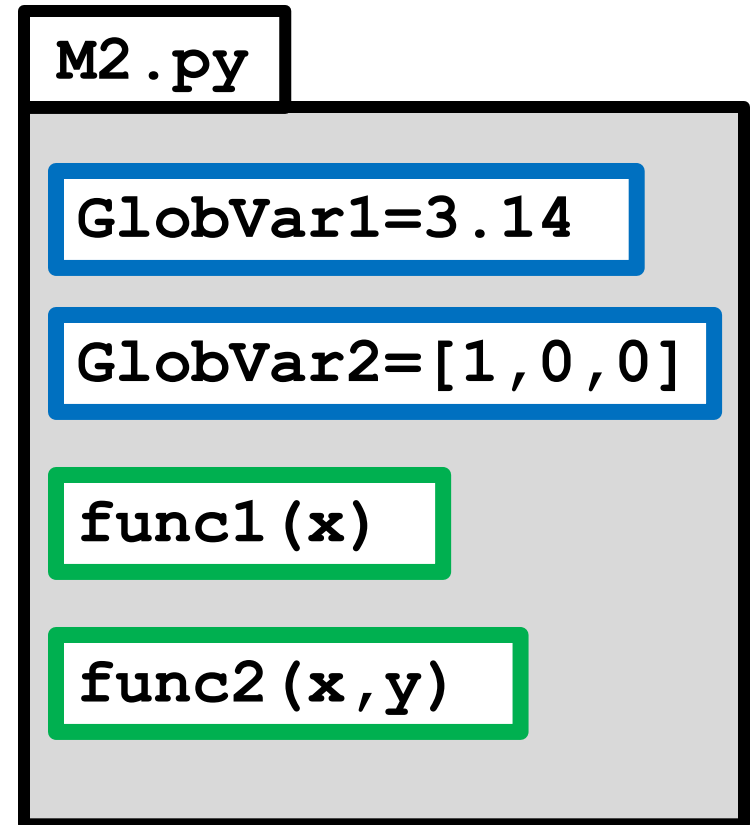
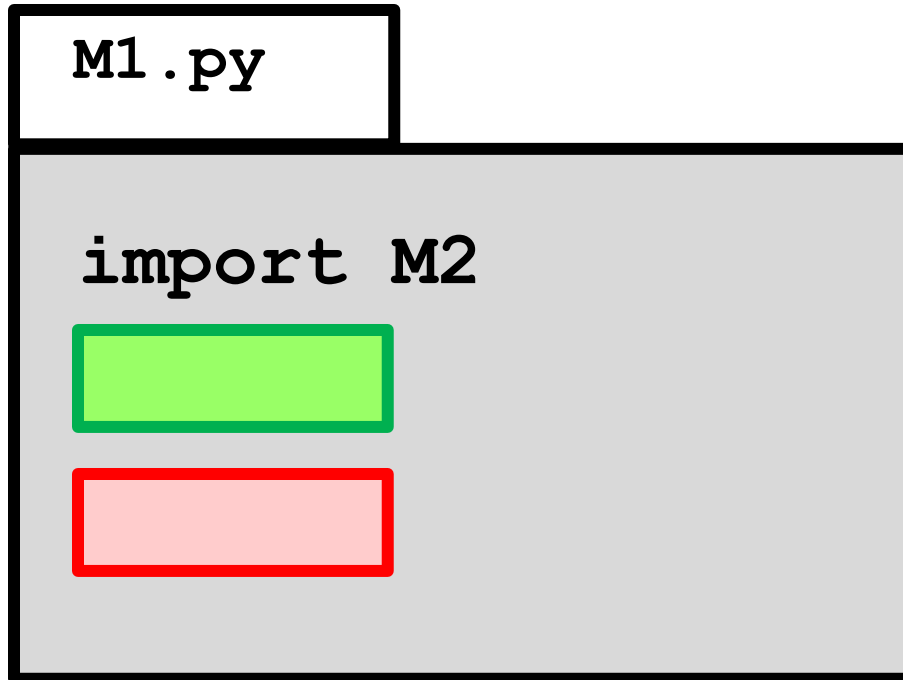
```
func1(x)
```


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

A diagram representing a Python file named M2.py. It has a title box at the top containing the text 'M2.py'. Below the title is a large grey rectangular area representing the file's content. The content is organized into four separate boxes with black outlines: a blue box containing 'GlobVar1=3.14', another blue box containing 'GlobVar2=[1,0,0]', a green box containing 'func1(x)', and a final green box containing 'func2(x,y)'. The blue boxes represent global variables, and the green boxes represent functions.

A function `[ ]` in `M1.py` could have a line like  
`a = M2.func2(x,M2.GlobVar1)`

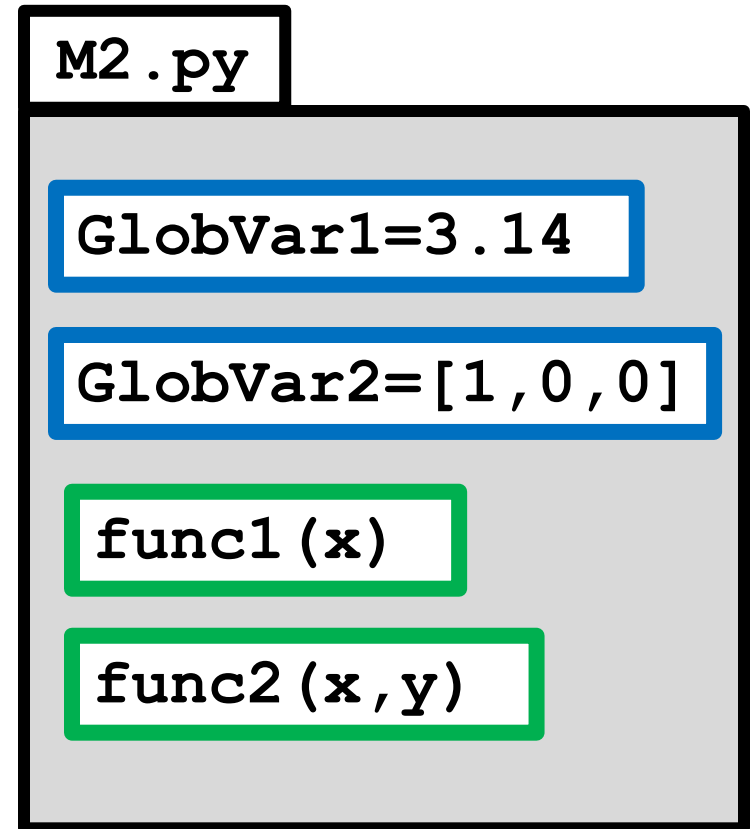
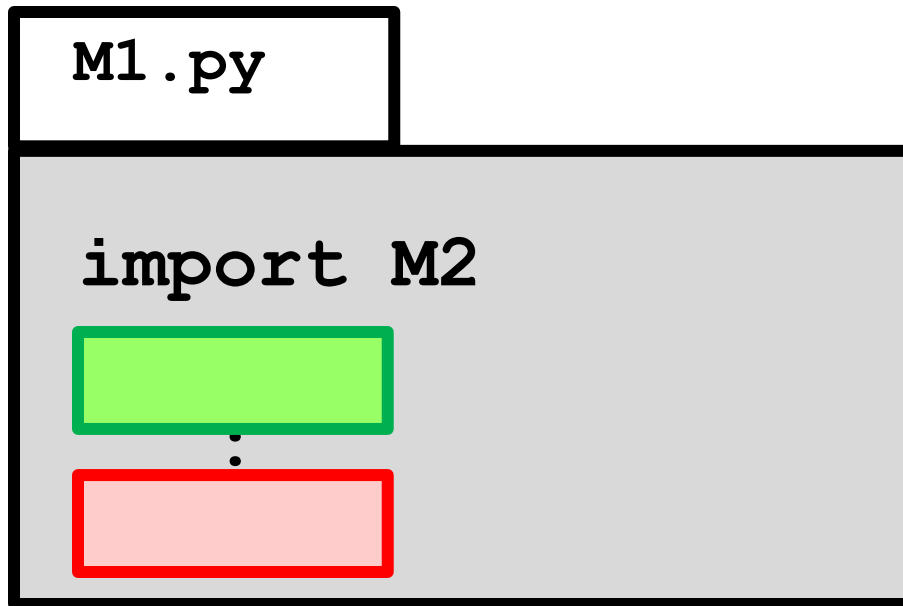
# What Does `import` Allow?



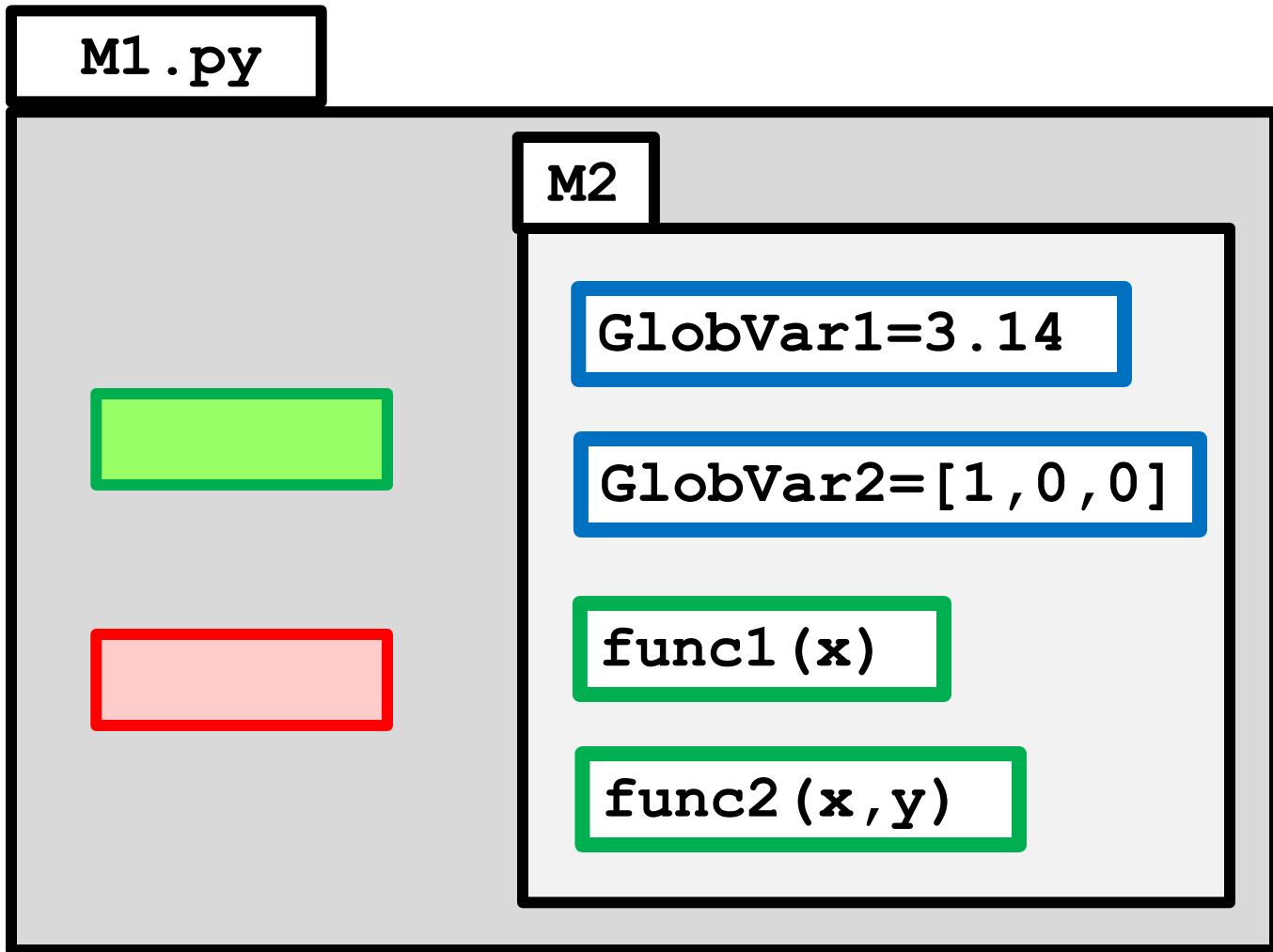
The script  in M1.py could have a line like

```
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?

```
M1.py
```

```
from M2 import*
```

```
[ ]
```

```
[ ]
```

```
M2.py
```

```
GlobalVar1=3.14
```

```
GlobalVar2=[1,0,0]
```

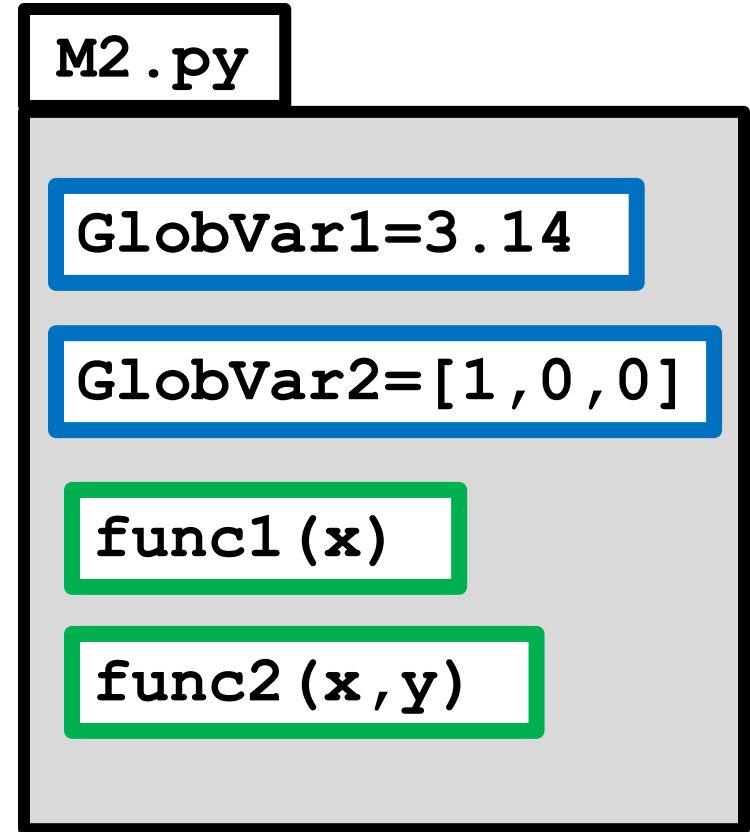
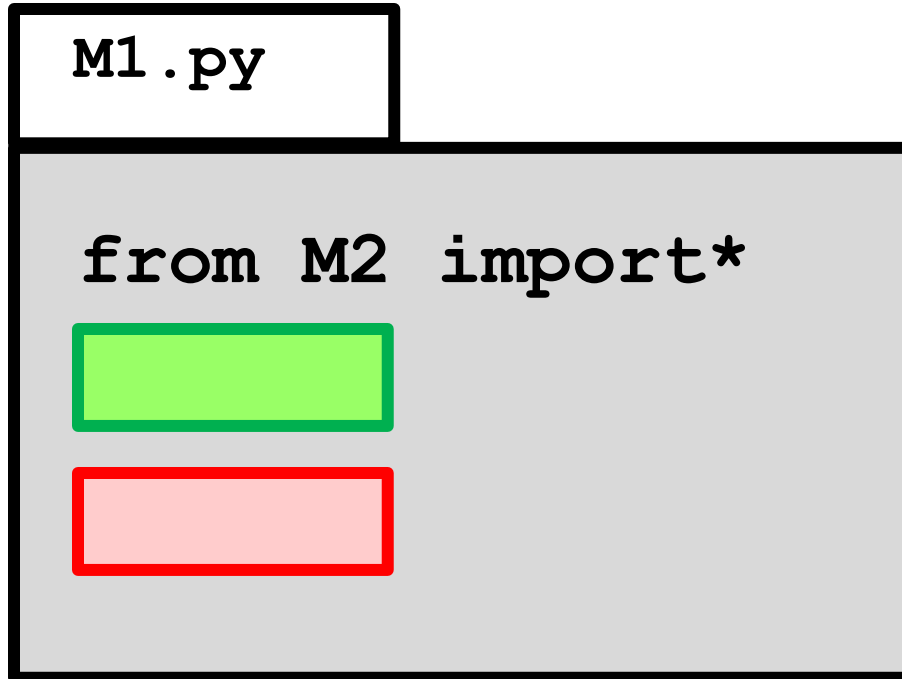
```
func1(x)
```


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

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

No dot notation

# What Does `import*` Allow?

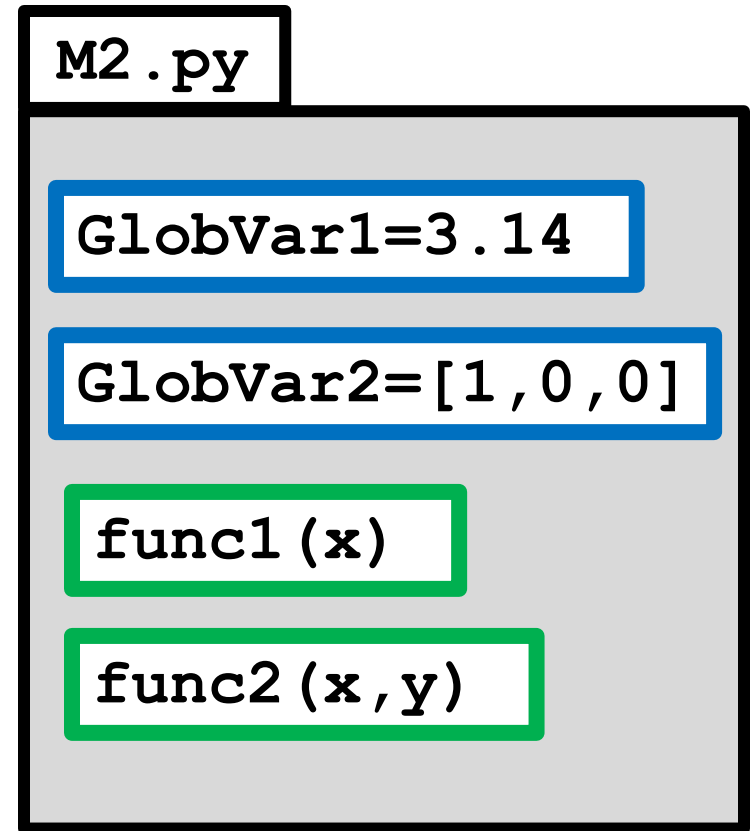
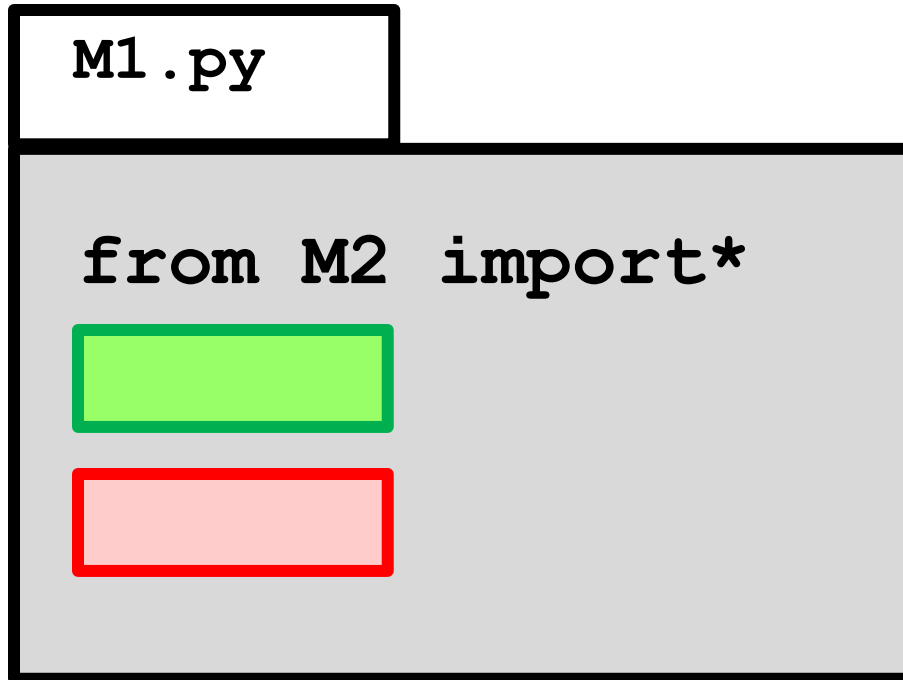


A script  in M1.py could have a line like `a = func2(x, GlobalVar2)`

No dot notation



# One way to Think about this...



is like this...

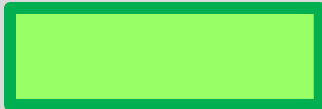
M1.py

```
GlobVar1=3.14
```

```
GlobVar2=[1,0,0]
```

```
func1(x)
```

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



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

# "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
```

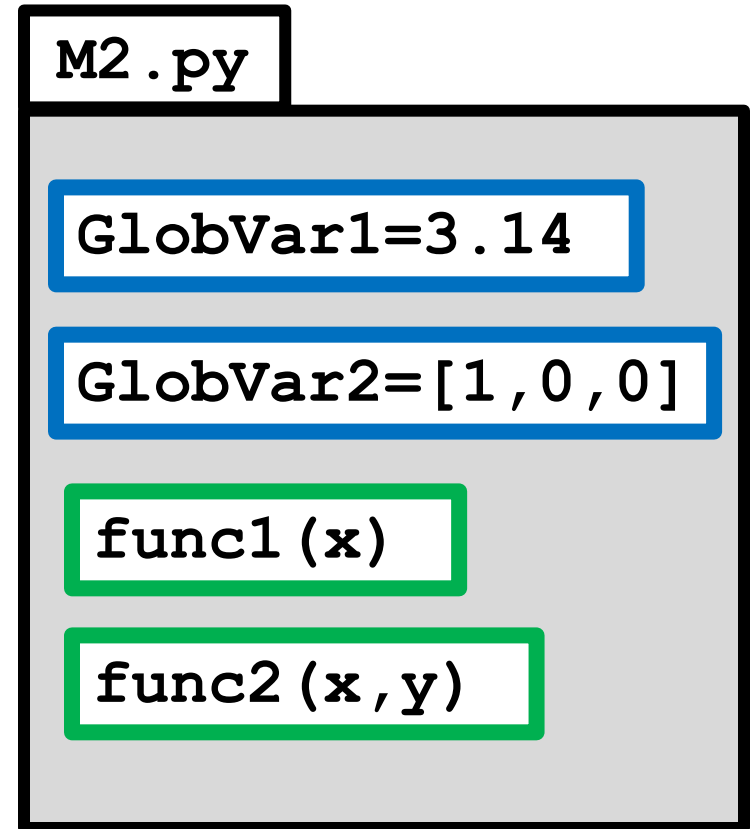
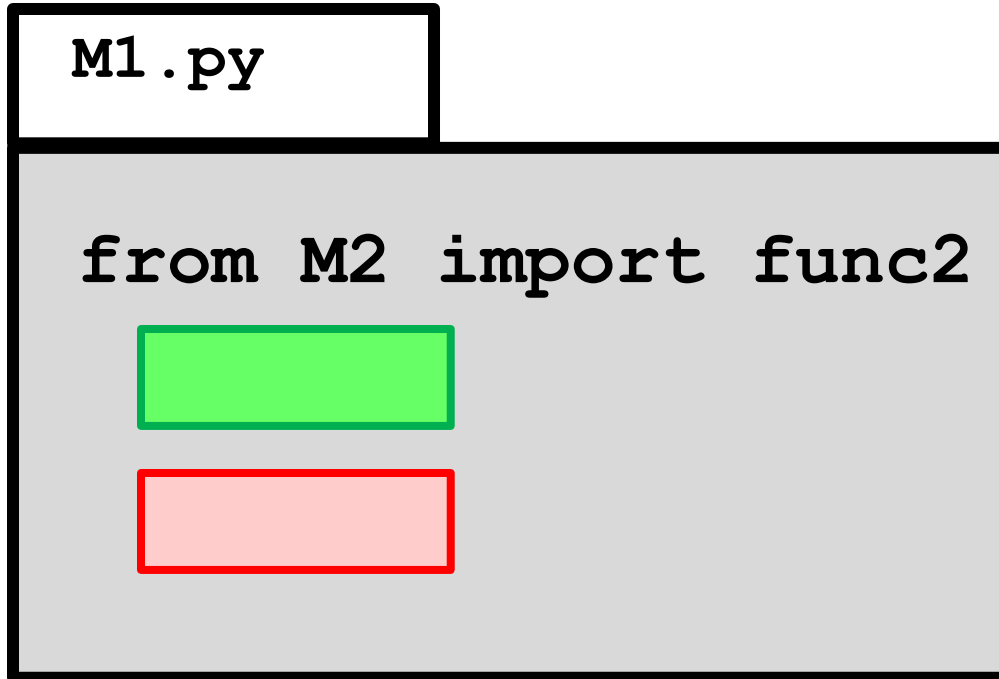
```
GlobVar2=[1,0,0]
```

```
func1(x)
```

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

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

# One way to think about this...



is like this...

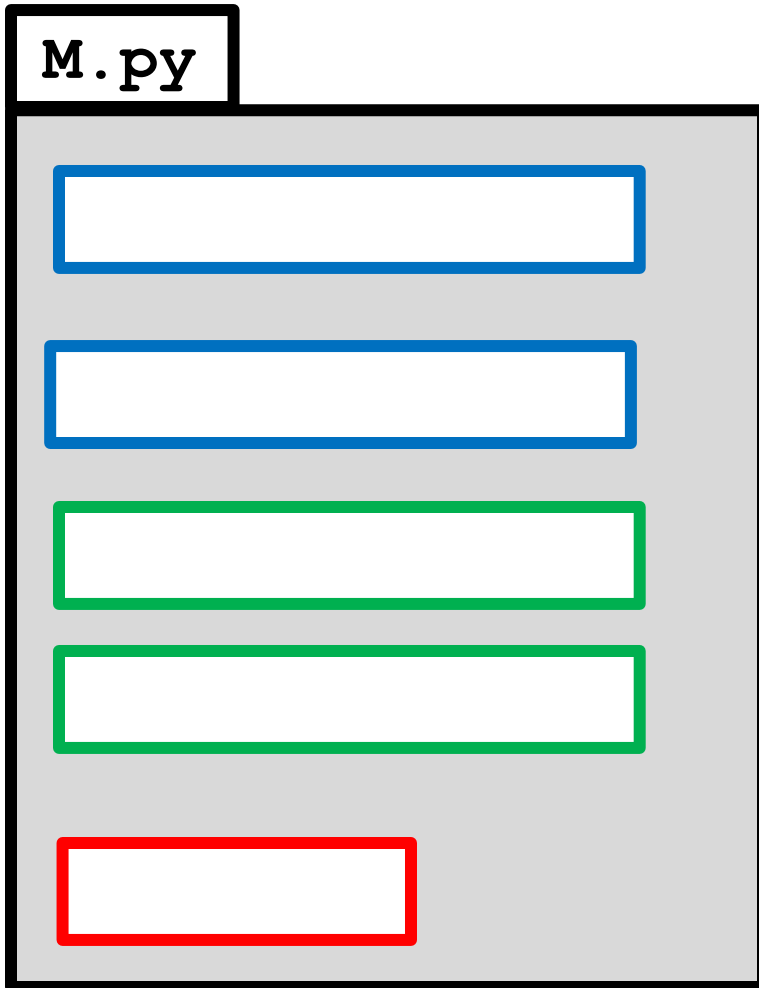
M1.py

func2(x, y)



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
```

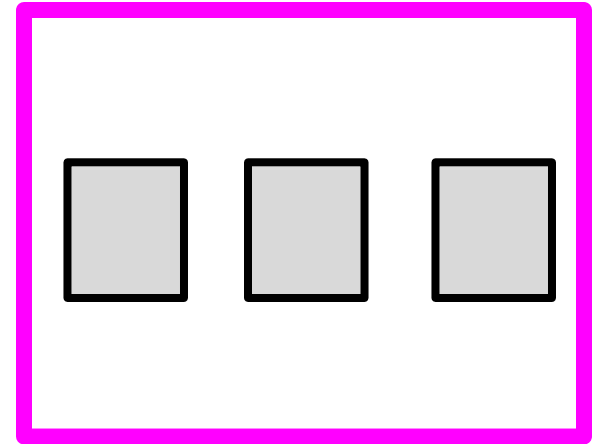
```
import M2  
:
```

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 >`

M1.py

```
import M2
```



M2.py



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

Result: the script in M1.py is executed.

M1.py

```
import M2
```



M2.py



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

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

M1.py

```
import M2
```

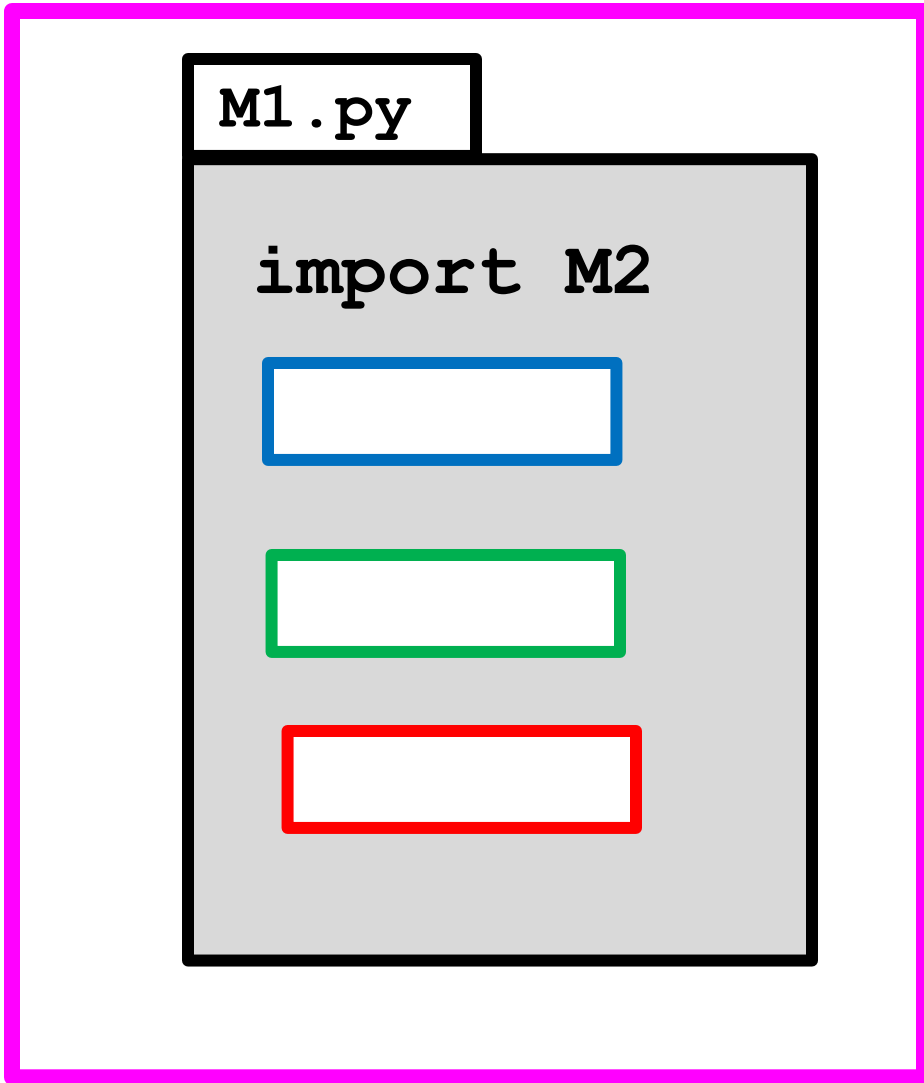


M2.py

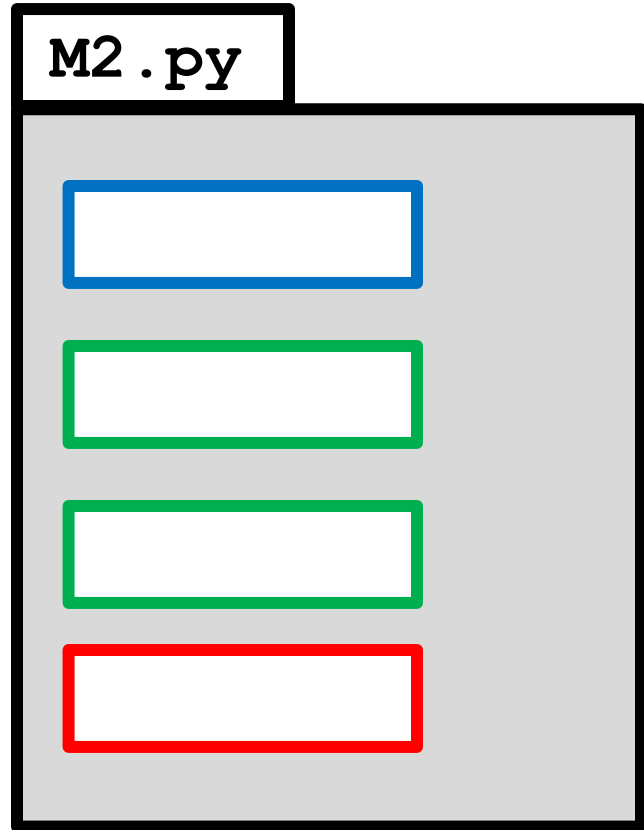


```
cwd > python M1.py
```

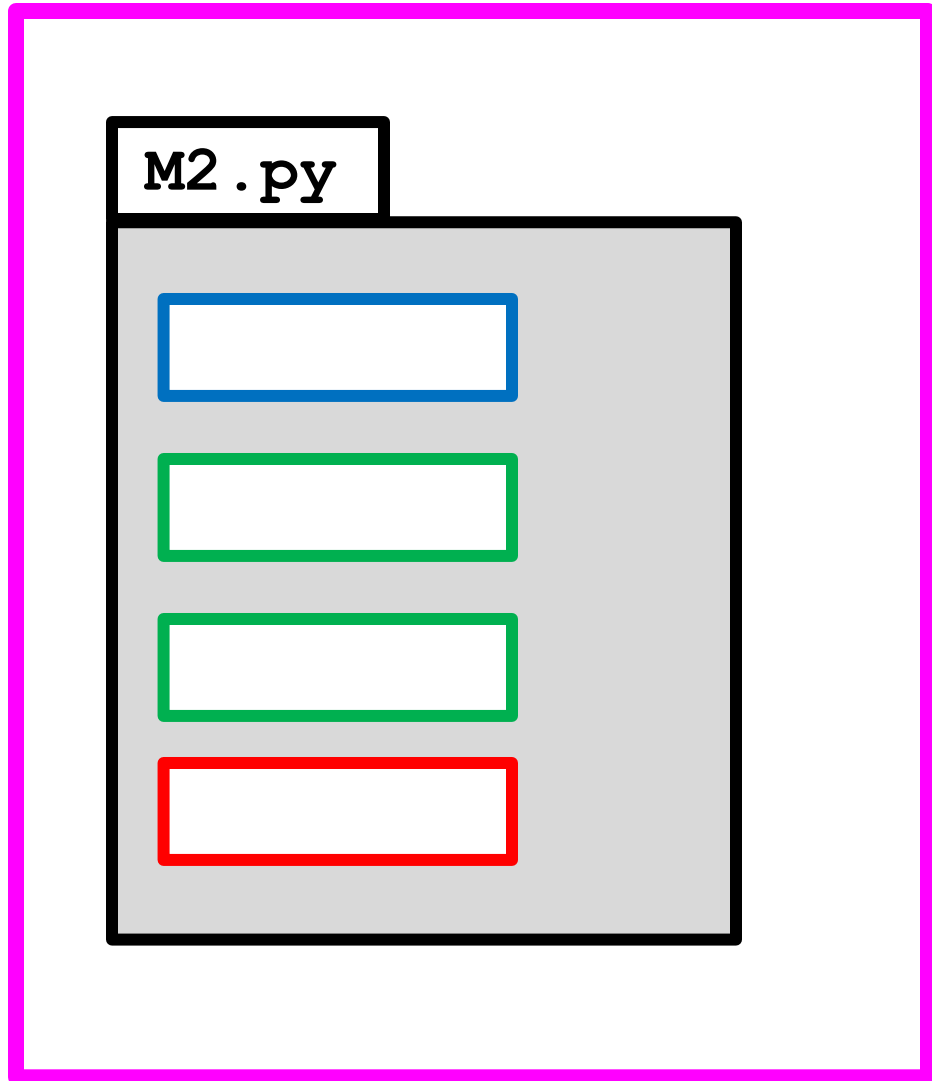
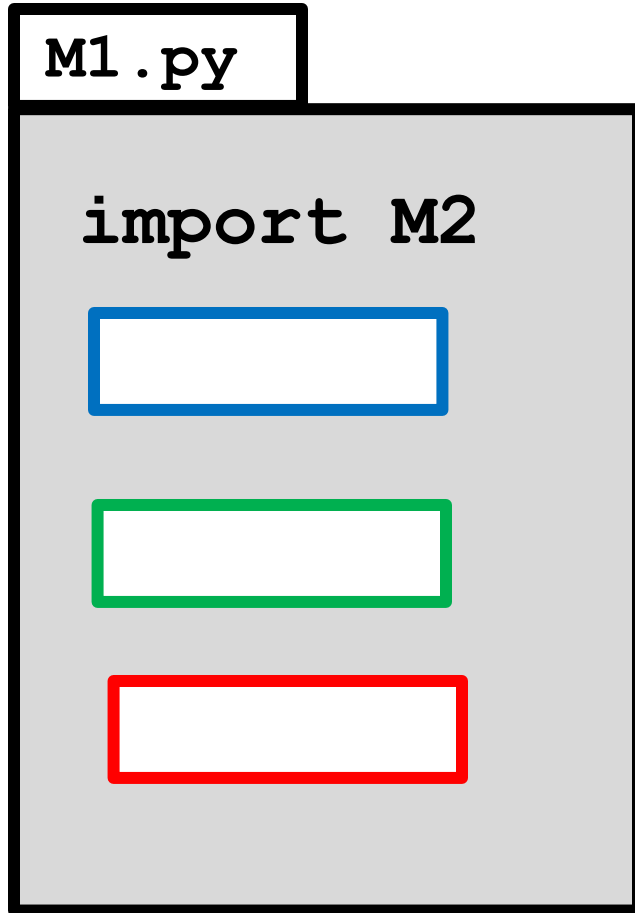
Result: Nothing happens because there is no script in M1.py to execute.



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



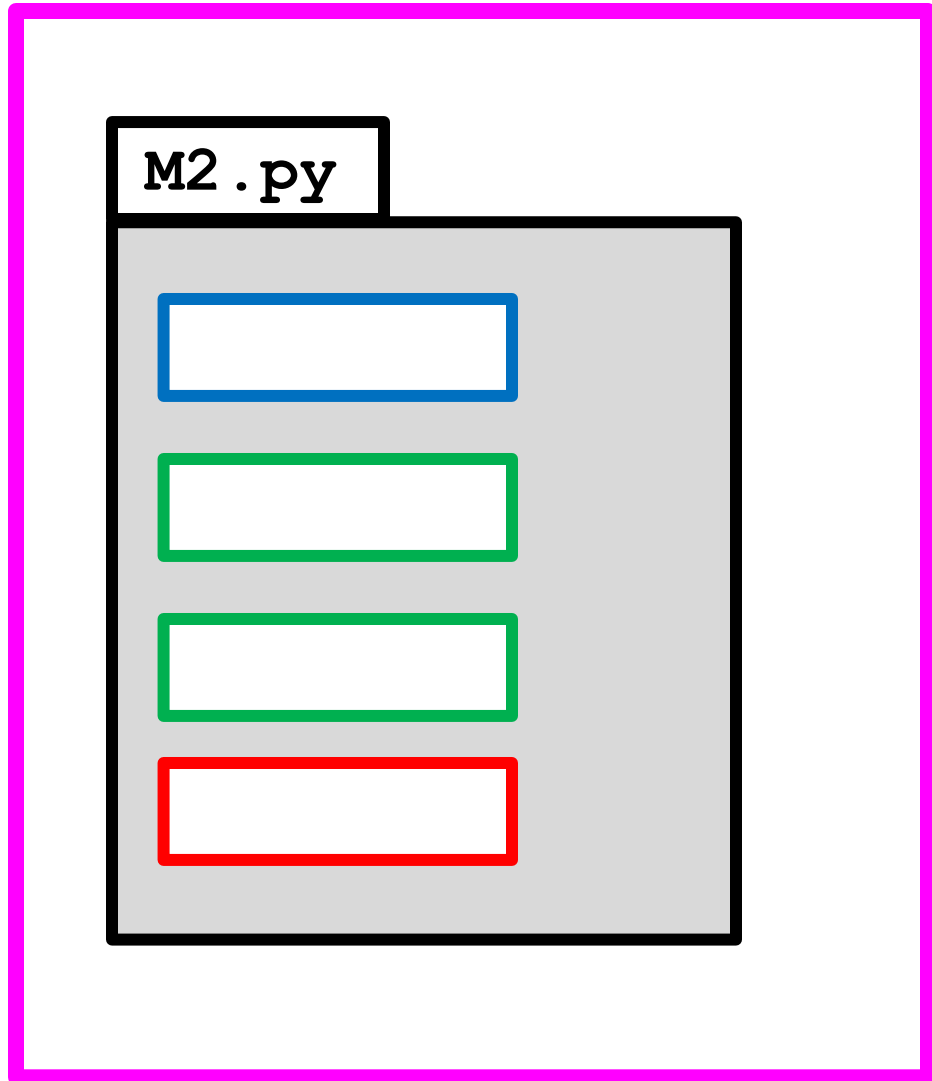
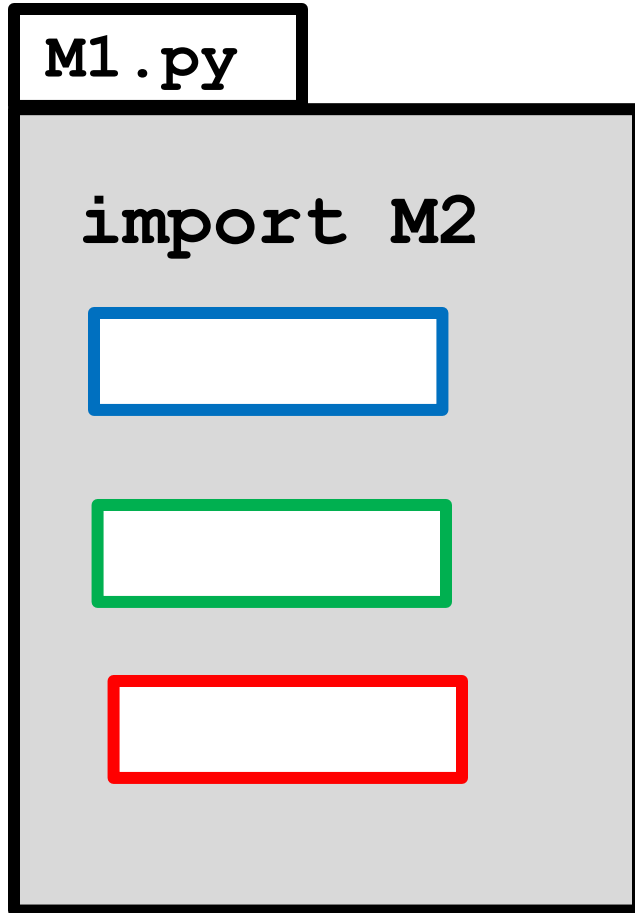
Error. Python cannot find M2



`cwd > python M1.py`

Error. Python cannot find M1





`cwd > python M2.py`

Fine. `M2.py` does not need `M1.py`

# 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.*

```
● 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.

A diagram illustrating the mapping of variables to memory. On the left, the variable names `s1`, `s2`, `x`, `y`, and `numMin` are listed vertically. To the right of each name is a horizontal double-headed arrow pointing to a light pink rectangular box, representing the memory location for that variable.

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

The diagram illustrates the state of variables in memory. On the left, the variable names `s1`, `s2`, `x`, `y`, and `numMin` are listed. On the right, there are five light red rectangular boxes representing memory locations. An arrow points from `s1` to the first box containing the string `'11:15'`. Arrows point from `s2`, `x`, `y`, and `numMin` to the subsequent four empty boxes, indicating that these variables have been allocated memory but have not yet been assigned a value.

```
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
```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	
<b>y</b>	⇒	
<b>numMin</b>	⇒	

## Function call

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

Let's track what happens...

```
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
```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	
<b>y</b>	⇒	
<b>numMin</b>	⇒	

## 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

```

```

s1  => '11:15'
s2  => '12:05'
x   => 
y   => 
numMin => 

```

```

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

```

```

s  => 
h  => 
m  => 
z  => 
return => 

```

The variable `s` is the function's parameter

```

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

```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	
<b>y</b>	⇒	
<b>numMin</b>	⇒	

<b>s</b>	⇒	
<b>h</b>	⇒	
<b>m</b>	⇒	
<b>z</b>	⇒	
<b>return</b>	⇒	

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

```

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

```

Diagram illustrating variable assignments:

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

```

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

```

Diagram illustrating function parameters and return value:

- s → [ ]
- h → [ ]
- m → [ ]
- z → [ ]
- return → [ ]

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

```
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
```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	
<b>y</b>	⇒	
<b>numMin</b>	⇒	

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

<b>s</b>	⇒	
<b>h</b>	⇒	
<b>m</b>	⇒	
<b>z</b>	⇒	
<b>return</b>	⇒	

Control passes from the red dot to the blue dot

```
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
```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	
<b>y</b>	⇒	
<b>numMin</b>	⇒	

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

<b>s</b>	⇒	'11:15'
<b>h</b>	⇒	
<b>m</b>	⇒	
<b>z</b>	⇒	
<b>return</b>	⇒	

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



```

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

```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	
<b>y</b>	⇒	
<b>numMin</b>	⇒	

```

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

```

<b>s</b>	⇒	'11:15'
<b>h</b>	⇒	
<b>m</b>	⇒	
<b>z</b>	⇒	
<b>return</b>	⇒	

We step through the function body, business as usual.

```

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

```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	
<b>y</b>	⇒	
<b>numMin</b>	⇒	

```

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

```

<b>s</b>	⇒	'11:15'
<b>h</b>	⇒	11
<b>m</b>	⇒	
<b>z</b>	⇒	
<b>return</b>	⇒	

We step through the function body, business as usual.

```

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

```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	
<b>y</b>	⇒	
<b>numMin</b>	⇒	

```

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

```

<b>s</b>	⇒	'11:15'
<b>h</b>	⇒	11
<b>m</b>	⇒	15
<b>z</b>	⇒	
<b>return</b>	⇒	

We step through the function body. Business as usual.

```

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

```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	
<b>y</b>	⇒	
<b>numMin</b>	⇒	

```

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

```

<b>s</b>	⇒	'11:15'
<b>h</b>	⇒	11
<b>m</b>	⇒	15
<b>z</b>	⇒	675
<b>return</b>	⇒	

We step through the function body. Business as usual.

```

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

```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	
<b>y</b>	⇒	
<b>numMin</b>	⇒	

```

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

```

<b>s</b>	⇒	'11:15'
<b>h</b>	⇒	11
<b>m</b>	⇒	15
<b>z</b>	⇒	675
<b>return</b>	⇒	675

We step through the function body. Business as usual.

```

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

```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	675
<b>y</b>	⇒	
<b>numMin</b>	⇒	

```

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

```

<b>s</b>	⇒	'11:15'
<b>h</b>	⇒	11
<b>m</b>	⇒	15
<b>z</b>	⇒	675
<b>return</b>	⇒	675

The return value is shipped back to the 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

```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	675
<b>y</b>	⇒	
<b>numMin</b>	⇒	

```

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

```

<b>s</b>	⇒	'11:15'
<b>h</b>	⇒	11
<b>m</b>	⇒	15
<b>z</b>	⇒	675
<b>return</b>	⇒	675

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

```
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
```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	675
<b>y</b>	⇒	
<b>numMin</b>	⇒	

Another  
function  
Call!

And the red dot moves to the next statement in the 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

```

```

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

```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	675
<b>y</b>	⇒	
<b>numMin</b>	⇒	

<b>s</b>	⇒	
<b>h</b>	⇒	
<b>m</b>	⇒	
<b>z</b>	⇒	
<b>return</b>	⇒	

We open up the Call Frame

```

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

```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	675
<b>y</b>	⇒	
<b>numMin</b>	⇒	

<b>s</b>	⇒	'12:05'
<b>h</b>	⇒	
<b>m</b>	⇒	
<b>z</b>	⇒	
<b>return</b>	⇒	

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

```

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

```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	675
<b>y</b>	⇒	
<b>numMin</b>	⇒	

```

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

```

<b>s</b>	⇒	'12:05'
<b>h</b>	⇒	12
<b>m</b>	⇒	
<b>z</b>	⇒	
<b>return</b>	⇒	

Execution of the function body starts.

```

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

```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	675
<b>y</b>	⇒	
<b>numMin</b>	⇒	

```

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

```

<b>s</b>	⇒	'12:05'
<b>h</b>	⇒	12
<b>m</b>	⇒	5
<b>z</b>	⇒	
<b>return</b>	⇒	

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

```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	675
<b>y</b>	⇒	
<b>numMin</b>	⇒	

```

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

```

<b>s</b>	⇒	'12:05'
<b>h</b>	⇒	12
<b>m</b>	⇒	5
<b>z</b>	⇒	5
<b>return</b>	⇒	

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

```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	675
<b>y</b>	⇒	
<b>numMin</b>	⇒	

```

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

```

<b>s</b>	⇒	'12:05'
<b>h</b>	⇒	12
<b>m</b>	⇒	5
<b>z</b>	⇒	5
<b>return</b>	⇒	5

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

```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	675
<b>y</b>	⇒	5
<b>numMin</b>	⇒	

```

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

```

<b>s</b>	⇒	'12:05'
<b>h</b>	⇒	12
<b>m</b>	⇒	5
<b>z</b>	⇒	5
<b>return</b>	⇒	5

That value is sent back to the red dot.

```
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
```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	675
<b>y</b>	⇒	5
<b>numMin</b>	⇒	50

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



```
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
```

<b>s1</b>	⇒	'11:15'
<b>s2</b>	⇒	'12:05'
<b>x</b>	⇒	675
<b>y</b>	⇒	5
<b>numMin</b>	⇒	50

# 50

The script is over. Global space disappears.

# 50

The script is over. *Global space disappears.*