SEE1002
Introduction to Computing for Energy and Environment

Part 3: Basic Python programming

Sec. 2: Structure of a Python program

Course Outline

Part 1: Introduction to computing

Part 2: Elements of Python programming

Section 1: Data and variables

Section 2: Elementary data structures

Section 3: Branching or decision making

Section 4: Loops

Section 5: Functions

Part 3: Basic Python programming

Section 1: Modules

Section 2: Structure of a Python program

Section 3: Good programming practices

Part 4: Python for science and engineering

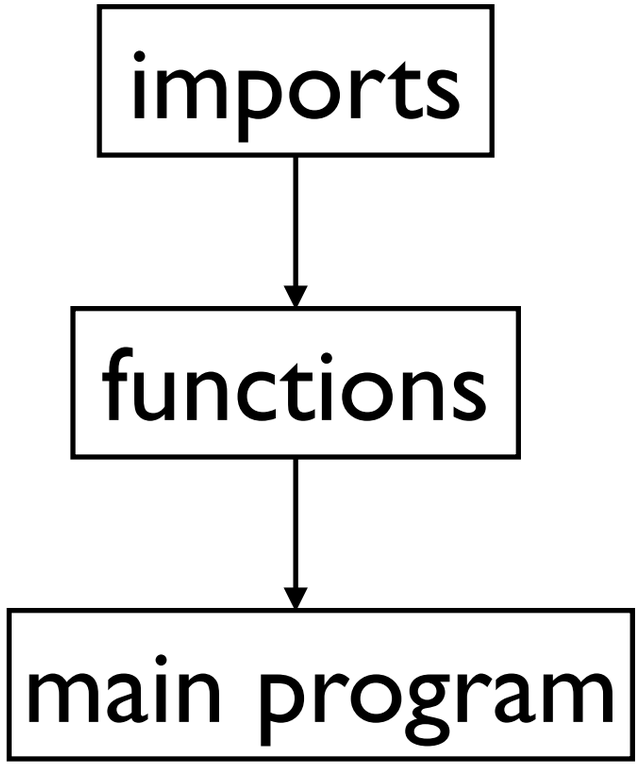
Section 1: File input and output

Section 2: Other topics

1. Motivation

How do we structure a program?

The standard way of structuring a Python program is as follows:



Rationale

Why is this order followed?

1. Modules must be imported before their contents are referenced.
2. Functions must be defined before they are called.

By putting everything at the beginning, we avoid potential problems.

Example 1a: standard order

```
from math import *

def areaRectangle(L,W):
    area=L*W
    return (area)

def areaCircle(r):
    area=pi*r**2
    return (area)

L = 1.0
W = 2.0
r = 1.0
print('The area of the rectangle =', areaRectangle(L,
print('The area of the circle =', areaCircle(r))
```

imports

functions

Main program

```
The area of the rectangle = 2.0
The area of the circle = 3.14159265359
```

works!

N.B. Program execution starts after function definitions.

Example 1b: undefined functions

```
from math import *

L = 1.0
W = 2.0
r = 1.0

print( 'The area of the rectangle =', areaRectangle(L,W) )
print( 'The area of the circle =', areaCircle(r) )

# defining functions at the end of the program is a bad idea!
def areaRectangle(L,W):
    area=L*W
    return area

def areaCircle(r):
    area=pi*r**2
    return area
```

imports

undefined

functions

should precede main program

```
    print( 'The area of the rectangle =', areaRectangle(L,W) )

NameError: name 'areaRectangle' is not defined
```

error!

N.B. Need to define functions before calling them.

Example 1c: unimported module

```
def areaRectangle(L,W):
    area=L*W
    return (area)

def areaCircle(r):
    area=pi*r**2
    return (area)


L = 1.0
W = 2.0
r = 1.0
print( 'The area of the rectangle =', areaRectangle(L,W) )
print( 'The area of the circle =', areaCircle(r) )

# importing module at the end is a bad idea!
from math import *
```

undefined

functions

imports

should precede functions and main program

```
    area=pi*r**2

NameError: name 'pi' is not defined
```

error!

2. Making the structure more explicit

Explicit `main()`

In the preceding examples, it's hard to distinguish the main program from the functions.

Although this is not strictly required, we can place the main program inside a function called `main`. Subsequently it can be called with `main()`.

```
from math import *

def areaRectangle(L,W):
    area=L*W
    return (area)

def areaCircle(r):
    area=pi*r**2
    return (area)

L = 1.0
W = 2.0
r = 1.0
print('The area of the rectangle =', areaRectangle(L,W))
print('The area of the circle =', areaCircle(r))
```

```
def main():
    statement1
    [...]
main()
```

Example 1d: explicit main

```
from math import *  # import has global scope

def areaRectangle(L,W):
    area=L*W
    return (area)

def areaCircle(r):
    area=pi*r**2
    return (area)

def main():  # this is our explicit main
    L = 1.0
    W = 2.0
    r = 1.0
    print( 'The area of the rectangle =', areaRectangle(L,W) )
    print( 'The area of the circle =', areaCircle(r) )

main()  # we need to call our main program in order for it to run
```

imports

functions

Main program *definition*

main program is easily visible

Main program *call*

```
The area of the rectangle = 2.0
The area of the circle = 3.14159265359
```

works!

What are the advantages of an explicit `main`?

Careful programmers tend to use an explicit `main`. Why?

1. Structure of the code is clearer.
2. Eliminates confusion and prevents accidents (forces variables in the main program to have local scope)

Alternative approach

Python programmers sometimes call the main program inside an `if` test:

```
from math import *  # import has global scope

def areaRectangle(L,W):
    area=L*W
    return (area)

def areaCircle(r):
    area=pi*r**2
    return (area)

def main():  # this is our explicit main
    L = 1.0
    W = 2.0
    r = 1.0
    print( 'The area of the rectangle =', areaRectangle(L,W) )
    print( 'The area of the circle =', areaCircle(r) )
    print(__name__)


if __name__ == "__main__":
    main()  # we need to call our main program in order for it to run
```

effective name of the file

```
The area of the rectangle = 2.0
The area of the circle = 3.141592653589793
__main__
```

Name of main program

Result is unchanged. This is equivalent to calling `main()` directly

`__name__`

The variable `__name__` returns the name of the Python program:

- `__main__` for a single program file
- The filename of the module when it's called inside a module.

Evaluating __name__ inside a module

```
from math import *

def areaRectangle(L,W):
    area=L*W
    print(__name__)
    return (area)

def areaCircle(r):
    area=pi*r**2
    return (area)
```

mylib.py

```
from mylib import *
L = 1.0
W = 2.0
r = 1.0
print( 'The area of the rectangle =', areaRectangle(L,W) )
print( 'The area of the circle =', areaCircle(r) )
```

main program

```
mylib
The area of the rectangle = 2.0
The area of the circle = 3.141592653589793
```

name of module

Why do programmers do this?

The `__name__` variable allows us to import a program file as a module! Without a `__name__` test the main program will be executed even if we only want to use some functions.

Module/program file with `__name__` test

```
from altmain import *
L = 1.0
W = 2.0
r = 1.0
print( 'The area of the rectangle =', areaRectangle(L,W)
print( 'The area of the circle =', areaCircle(r) )
```

```
The area of the rectangle = 2.0
The area of the circle = 3.14159265359
```

Output is as expected

Program file without `__name__` test

```
from example1d import *
L = 1.0
W = 2.0
r = 1.0
print( 'The area of the rectangle =', areaRectangle(L,W) )
print( 'The area of the circle =', areaCircle(r) )
```

```
The area of the rectangle = 2.0
The area of the circle = 3.141592653589793
The area of the rectangle = 2.0
The area of the circle = 3.141592653589793
```

Output is doubled!

Comparison of program files

with `__main__` test

```
from math import *  # import has global scope

def areaRectangle(L,W):
    area=L*W
    return (area)

def areaCircle(r):
    area=pi*r**2
    return (area)

def main():  # this is our explicit main
    L = 1.0
    W = 2.0
    r = 1.0
    print( 'The area of the rectangle =', areaRectangle(L,W) )
    print( 'The area of the circle =', areaCircle(r) )


if __name__ == "__main__":
    main()  # we need to call our main program in order for it to
```

`main` is only executed when the file is run

without `__main__` test

```
def areaRectangle(L,W):
    area=L*W
    return (area)

def areaCircle(r):
    area=pi*r**2
    return (area)

def main():  # this is our explicit main
    L = 1.0
    W = 2.0
    r = 1.0
    print( 'The area of the rectangle =', areaRectangle(L,W) )
    print( 'The area of the circle =', areaCircle(r) )


main()  # we need to call our main program in order for it to r
```

`main` is executed even when only the functions inside a module are used

3. Local and global scope revisited

Global scope

Previously we mentioned that variables defined within functions have local scope, i.e., they are local variables.

- To keep the discussion simple, we didn't mention that variables declared outside of functions have global scope. *However, they aren't truly global variables because they cannot be modified.*
- What this means is that if we don't put our main program inside a function, then **its variables can be accessed inside all other functions** even if they are not passed as arguments.
- In other languages, global scope *can lead to unexpected results! However, Python keeps us from causing accidents!*

Example 2a: global scope with implicit `main`

```
from math import *

def areaRectangle(L,W):
    '''
    calculate area of rectangle
    input: L (length), W (width)
    output: area
    '''
    area=L*W
    return (area)

def areaCircle(r):
    '''
    calculate area of circle
    input: r (radius)
    output: area
    '''
    area=pi*r**2
    print( 'L=', L ) # we can access L and W without passing them
    print( 'W=', W )
    return (area)

# implicit main
L = 1.0   # L and W are global variables
W = 2.0
r = 1.0
print( 'The area of the rectangle =', areaRectangle(L,W) )
print( 'The area of the circle =', areaCircle(r) )
```

L and W do not appear as arguments

implicit main

```
The area of the rectangle = 2.0
The area of the circle = L= 1.0
W= 2.0
3.14159265359
```

no problems

Example 2b: local scope with explicit `main`

```
def areaRectangle(L,W):
    '''
    calculate area of rectangle
    input: L (length), W (width)
    output: area
    '''
    area=L*W
    return (area)

def areaCircle(r):
    '''
    calculate area of circle
    input: r (radius)
    output: area
    '''
    area=pi*r**2
    print( 'L=', L )
    print( 'W=', W )
    return (area)

def main():
    L = 1.0 # L and w are local variables
    W = 2.0
    r = 1.0
    print( 'The area of the rectangle =', areaRectangle(L,W) )
    print( 'The area of the circle =', areaCircle(r) )


main()
```

L and W must be passed as arguments!

explicit main

```
    print( 'L=', L ) # we can access L and W without passing them

NameError: name 'L' is not defined
```

error!

Modifying global variables

- In other languages global variables can be modified within functions. *This is very dangerous!*
- Python allows us to access global variables, but it tries to prevent us from modifying them. If one tries to do an assignment the variable will be interpreted as a local variable. *This is very helpful!*

Example 2c: attempted modification of global variable

```
from math import *

def areaRectangle(L,W):
    area=L*W
    return area

def areaCircle(r):
    area=pi*r**2
    print('L=', L)
    print( 'W=', W )
    L=1.0    # we aren't allowed to modify a global variable
    return (area)

L = 1.0 # L and W are global
W = 2.0
r = 1.0
print( 'The area of the rectangle =', areaRectangle(L,W) )
print( 'The area of the circle =', areaCircle(r) )
```

attempted modification

```
    print('L=', L)

UnboundLocalError: local variable 'L' referenced before assignment
```

error!

Example 3a: access to global variable

x is a global variable

```
def function():
    print('inside function: x=', x)

# implicit main starts here
x=1 # x is a global variable
print('original value of x before fn call =',x)
function()
print( 'original value of x after fn call=',x )
```

```
original value of x before fn call = 1
inside function: x= 1
original value of x after fn call= 1
```

Global variable can be accessed everywhere.

Example 3b: local variable

local

```
def function():
    x=2 # local assignment creates a local variable
    print('inside function: x=', x)

# implicit main
x=1 # this looks like a global variable...
print( 'original value of x before fn call =',x )
function()
print( 'original value of x after fn call=',x)
```

same variable names

global

```
original value of x before fn call = 1
inside function: x= 2
original value of x after fn call= 1
```

Assignment inside a function turns x into a local variable

Example 3c: attempted modification of global variable

```
def function():
   print( 'inside function: x=',x)
   x=2 # turns x into local variable everywhere in function

x=1
print( 'original value of x before fn call =',x)
function()
print( 'original value of x after fn call=',x)
```

local

```
original value of x before fn call = 1
Traceback (most recent call last):
```

```
    print( 'inside function: x=',x)

UnboundLocalError: local variable 'x' referenced before assignment
```

Assignment turns x into a local variable within the function, but a variable can't be used before it's assigned!

Summary

1. For simple one-file programs, ordering the code with imports, functions and main program avoids problems.
2. An explicit `main` can be used to make the structure of the code clearer and avoid problems.
3. The main program file can be used as a module if `main` is called inside of `if __name__`.
4. Variables declared inside an implicit main have global scope, but they cannot be modified inside functions.