

Module

- Collection of things
 - function definition, classes, variables, and executable statements
- executed when module is imported
- modules have private symbol tables
- avoids name clash for global variables
- accessible as *module.globalname*
- file name is *module name + .py*

Module

- Collection of stuff in *foo.py* file
 - functions, classes, variables
- Importing modules:
 - `import time`
`time.sleep(3)`
 - `from time import sleep`
`sleep(3)`
- Can import all names defined by module
 - `from graphics import *`

Module

Import with rename:

- `import graphics as g`

`g.GraphWin("test", 500, 500)`

- `from time import sleep as delay`

`delay(3)`

Packages_[P/Q/M]

Collection of modules in directory

- Each (sub)package is represented as directory.
- Must have `__init__.py` file which can be empty
- May contain subpackages

Directory structure under package.P.Q.M_without_imports

```
. -> test.py    P
./P ->          __init__.py  Q
./P/Q ->        __init__.py  M
./P/Q/M ->      __init__.py  Foo.py
```

#contents of Foo.py

```
def foo():
    print 'Hi, there!'
```

```
if __name__ == '__main__':
    foo()
```

#contents of __init__.py

#test cases

- `from P.Q.M.Foo import foo`
`foo()`
- `import P.Q.M.Foo`
`P.Q.M.Foo.foo()`
- `import P.Q.M.Foo as MyFoo`
`MyFoo.foo()`
- `from P.Q.M import Foo`
`Foo.foo()`
- `from P.Q import M`
`M.Foo.foo()` *#Foo?*
- `from P import Q`
`Q.M.Foo.foo()` *#M?*
- `import P`
`P.Q.M.Foo.foo()` *#Q?*

Error: do not work

Lambda Functions



- small **anonymous functions created on the fly**
- mainly used in combination with the higher order functions filter(), map() and reduce().
- Lambda argument_list: expression

lambda x,y : x+y

lambda vrs. def



```
def add(x, y):  
    return x + y
```

```
Ladd = lambda x, y: x+y
```

Why use lambda functions?



When using verbose function declaration it is often the case that the function's verbose declaration can be verbose, even for functions that don't require such verbosity.

Verbose

```
def ispos(n):  
    return n > 0  
b = filter(ispos, aList)
```

```
b = []  
for a in aList:  
    if a > 0:  
        b.append(a)
```

Vs.

```
b = filter(lambda n: n > 0, aList)
```

Not Verbose

Also, there are some valid concerns about namespace clutter and verbosity.

Lambda Functions (map)



Map (function, sequence)

- function is applied on each item in the sequence
- A newly formed list is returned

```
a = [1,2,3,4]
```

```
b = [17,12,11,10]
```

```
print map(lambda x,y:x+y, a,b)
```


Lambda Functions(filter)



filter(function, sequence)

- function returns a boolean as filter criteria
- a newly formed list is returned

```
fib = [0,1,1,2,3,5,8,13,21,34,55]
```

```
print filter(lambda x: x % 2, fib)
```

```
print filter(lambda x: x % 2 == 0, fib)
```

Lambda Functions_(reduce)



reduce(function, sequence)

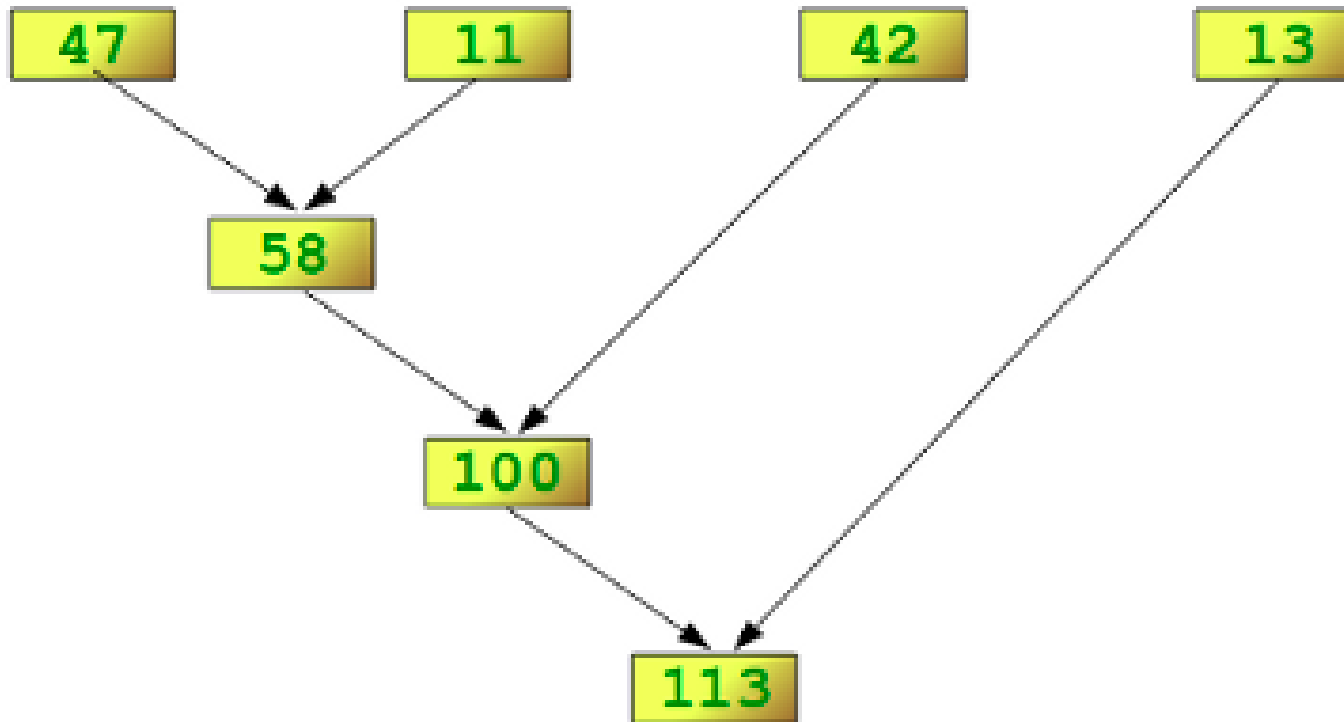
- return a single value
- call binary function on the first two items
then on the result and next item
- iterates

```
print reduce(lambda a,b: a if (a > b) else b,  
             [47,11,42,102,13])
```

Lambda Functions_(reduce)



```
print reduce(lambda x,y: x+y, [47,11,42,13])
```



List Comprehension

List Comprehensions allow us to do all sorts of things:

- Single-function single-line code
- Apply a function to each item of an iterable
- Filter using conditionals
- Cleanly nest loops

List Comprehension

To double the values in a list and assign to a new variable:

```
winning_lottery_numbers = [0, 4, 3, 2, 3, 1]
fake_lottery_numbers = []

for number in winning_lottery_numbers:
    fake_lottery_numbers.append(2 * number)

fake_lottery_numbers =
[2*n for n in winning_lottery_numbers]
```

List Comprehension

Create lists without `map`, `filter`, `lambda`

Syntax:

- expression followed by for clause
- zero or more for or of clauses

```
vec = [2,4,6]
```

```
print [3*x for x in vec]          #[6, 12, 18]
```

```
print [{x: x**2} for x in vec]    #[{2: 4}, {4: 16}, {6: 36}]
```

List Comprehension(cross product)

```
vec1 = [2,4,6]
```

```
Vec2 = [4,3,-9]
```

```
#[8,6,-18, 16,12,-36, 24,18,-54]
```

```
print [x*y for x in vec1 for y in vec2]
```

```
#[8,12,-54]
```

```
print [vec1[i]*vec2[i] for i in range(len(vec1))]
```

List Comprehension(condition-filtering)

Syntax:

```
[<expression> for <value> in <collection> if <condition>]
```

```
vec = [2,4,6]
```

```
print [3*x for x in vec if x > 3]    #[12, 18]
```

```
print [3*x for x in vec if x < 2]    #[]
```