Programming in Python

Lecture 4

Functions & recursion

Functions & recursion. PEP-8.

1. Functions.
2. Scope of the name’s visibility.
3. Arguments.
4. Function Design Concepts.
5. Recursive Functions.
6. Anonymous Functions: lambda.

1.Functions

Functions are a nearly universal program-structuring device. Functions serve two primary development roles:

- **Maximizing code reuse and minimizing redundancy**

As in most programming languages, Python functions are **the simplest way to package logic you may wish to use in more than one place and more than one time**. Up until now, all the code we've been writing has run immediately. **Functions allow us to group and generalize code to be used arbitrarily many times later.** They allow us to reduce code redundancy in our programs, and thereby **reduce maintenance effort**.

- **Procedural decomposition**

Functions also provide a tool for splitting systems into pieces that have well-defined roles. It's easier to implement the smaller tasks in isolation than it is to implement the entire process at once.

1.1.Creation

The ***def*** statement creates a function object and assigns it to a name (but does not call it):

def <name>(arg1, arg2,... ,argN =value):
<statements>
…
return <value>

The function does not exist until the interpreter reaches the def statement and executes it. It is possible (and sometimes even useful) to nest ***def*** statements inside **if** statements, ***while*** loops, and even other ***def*** statements. Most often, ***def*** statements are inserted into module files and generate functions when executed during the first ***import*** operation.

As with any other assignment operation, **the function name becomes a reference to the function object**. A function, like other objects, can be associated with several names, can be stored in a list, etc.

The argument names in the header line will be associated with the objects passed to the function at the call point.

The ***return*** statement can be located anywhere in the body of the function - **it terminates the function and passes the result to the calling program**. It is optional - if it is absent, the function ends when the control flow reaches the end of the function body. From a technical point of view, a function without a ***return*** statement automatically returns a ***None*** object, however this value is usually ignored.

Functions are ordinary objects; they are explicitly written to memory during program execution. A function name is a reference to an object.

```
def func(): ... # Creates a function object
othername = func # Assign function object
othername() # Call func
```

Polymorphism example:

```
def times(x, y):  # Creates a function  and assign it a name
        return x * y  # function body
>>> times(2, 4) # Function call. Arguments are numbers.
8
>>> times('Ni', 4) # Function call. Arguments are a string and a number.
'NiNiNiNi'
```

This is the most important difference between the philosophy of the Python language and the programming languages with static typing (C ++, Java): **the program code does not make assumptions about data types**. When programming in Python, **object interfaces**, not data types, **are considered**.

1.2. An example

- The intersection of two sequences:
- *def intersect(seq1, seq2):*

```
    res = []  # Initially empty result
    for x in seq1: # Sequence seq1 traversal
        if x in seq2: # Common element?
            res.append(x) # Add to end
    return res
```

```
>>> s1 = “SPAM”
>>> s2 = “SCAM”
>>> intersect(s1, s2) # Strings
[‘S’, ‘A’, ‘M’]
```

- The first parameter must have support for ***for*** loops, and the second should support the ***in*** operator, which performs an entry check. **Any two objects that meet these conditions will be processed regardless of their types**:

```
>>> x = intersect([1, 2, 3], (1, 4)) # Mixing types (list and tuple)
>>> x # Object with result
[1]
```

- This function can be replaced with a single list generator expression that demonstrates a classic example of a data fetch cycle:

```
>>> [x for x in s1 if x in s2]
[‘S’, ‘A’, ‘M’]
```

2. Scope of the name's visibility

When you use a name in a program, Python creates, changes, or looks up the name in what is known as **a namespace—a place where names live**. When we talk about the search for a name's value in relation to code**, the term *scope* refers to a namespace**: that is, **the location of a name's assignment in your source code determines the scope of the name's visibility to your code.**

Because names are not declared ahead of time, Python uses the location of the assignment of a name to associate it with (i.e., *bind* it to) a particular namespace. In other words, **the place where you assign a name in your source code determines the namespace it will live in**, and hence its **scope of visibility**.

Besides packaging code for reuse, functions add an extra namespace layer to your programs to minimize the potential for collisions among variables of the same name—***by default, all names assigned inside a function are associated with that function's namespace, and no other*.** This rule means that:

- **Names assigned inside a *def* can only be seen by the code within that *def***. You cannot even refer to such names from outside the function.
- **Names assigned inside a *def* do not conflict with variables outside the *def***, even if the same names are used elsewhere. A name ***X*** assigned outside a given ***def*** (i.e., in a different ***def*** or at the top level of a module file) is a completely different variable from a name ***X*** assigned inside that ***def***.

2.1. Local, nonlocal and global scope

Variables may be assigned in three different places, corresponding to three different scopes:

- If a variable is assigned inside a ***def***, it is ***local*** to that function.
- If a variable is assigned in an enclosing def, it is ***nonlocal*** to nested functions.
- If a variable is assigned outside all ***defs***, it is ***global*** to the entire file.

We call this ***lexical scoping*** because variable scopes are determined entirely by the locations of the variables in the source code of your program files, not by function calls.

Example:

```
X = 99 # Global (module) scope X
def func():
    X = 88 # Local (function) scope X: a different variable
```

Even though both variables are named X, their scopes make them different.

The **net effect** is that function scopes help to avoid name clashes in your programs and help to make functions more self-contained program units—their code need not be concerned with names used elsewhere.

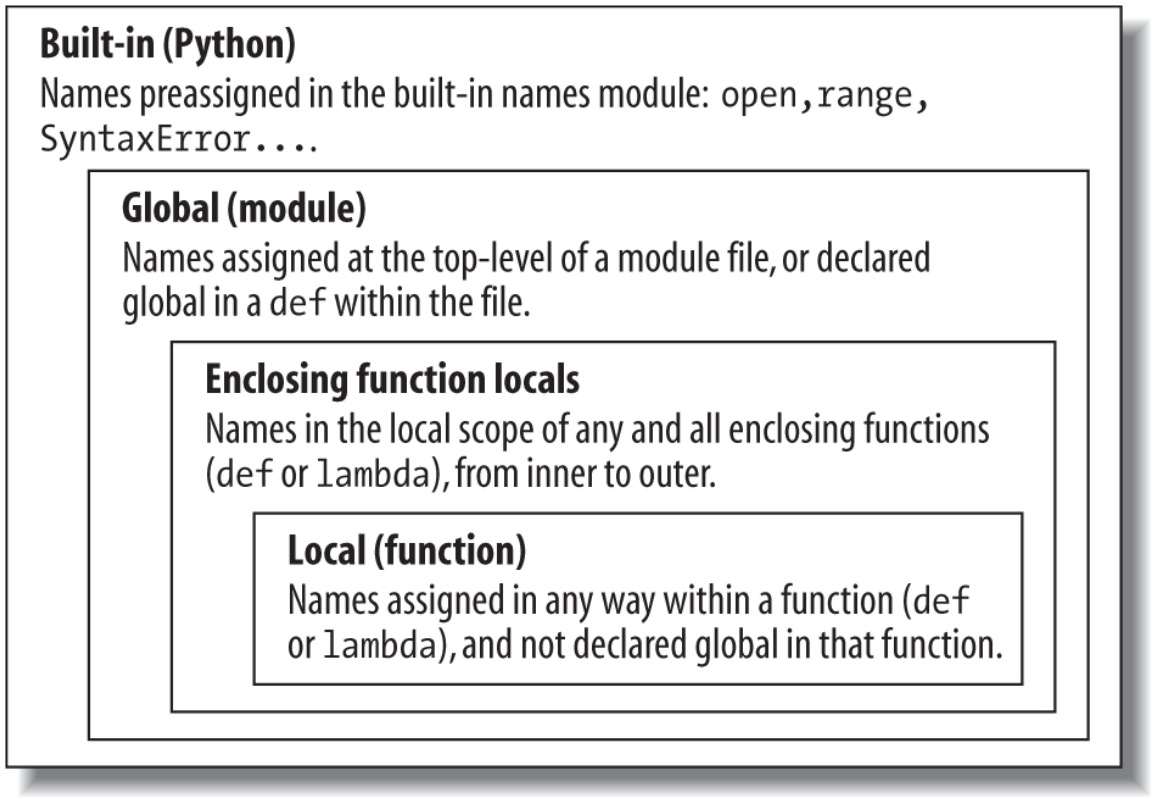
Local, nonlocal and global scope

Variables Functions define a ***local scope*** and modules define a ***global scope*** with the following properties:

- **The enclosing module is a global scope**. Each module is a global scope—that is, a namespace in which variables created (assigned) at the top level of the module file live. **Global variables become attributes of a module object** to the outside world after imports but can also be used as simple variables within the module file itself.
- **The global scope spans a single file only**. There is really no notion of a single, all-encompassing global file-based scope in Python. Instead, names are partitioned into modules, and you must always import a module explicitly if you want to be able to use the names its file defines. When you hear "global" in Python, think "module."
- **Assigned names are local unless declared global or nonlocal**. By default, all the names assigned inside a function definition are put in the local scope (the namespace associated with the function call). If you need to assign a name that lives at the top level of the module enclosing the function, you can do so by declaring it in a ***global*** statement inside the function. If you need to assign a name that lives in an enclosing ***def***, you can do so by declaring it in a ***nonlocal*** statement.
- **All other names are enclosing function local, global, or built-in**. Names not assigned a value in the function definition are assumed to be *enclosing* scope local, defined in a physically surrounding def statement; *global* that live in the enclosing module's namespace; or *built-in* in the predefined built-in module Python provides.
- **Each call to a function creates a new local scope**. Every time you call a function, you create a new local scope—that is, a namespace in which the names created inside that function will usually live.

2.2. Name Resolution: The LEGB Rule.

- **When you use an unqualified name inside a function, Python searches up to four scopes**—the local (***L***) scope, then the local scopes of any enclosing (***E***) defs and lambdas, then the global (***G***) scope, and then the built-in (***B***) scope—and stops at the first place the name is found. If the name is not found during this search, Python reports an error.
- **When you assign a name in a function** (instead of just referring to it in an expression), **Python always creates or changes the name in the local scope**, unless it's declared to be global or nonlocal in that function.
- When you assign a name outside any function (i.e., at the top level of a module file), the local scope is the same as the global scope — the module's namespace.



2.3. An example

```
# Global scope: X and  func are assigned in the module
X = 99
def func(Y):
        # local scope: Y and Z are assigned in the function
        Z = X + Y # X – global variable
        return Z
func(1) # func in the module: returns 100
```

- Global names: *X* and *func.*
- Local names: *Y* and *Z*.
- **The idea of this name's separation is that local variables play the role of temporary names**, which are necessary only during the function run. So, the argument Y and the result of adding Z exist only inside the function - these names do not intersect with the enclosing namespace of the module (or with the scopes of any other functions).
- Separation of names into global and local also facilitates the understanding of functions, since most of the names used in a function appear directly in the function itself, and not in some other place inside the module. In addition, you can be sure that the local names will not be changed by any other remote function in the program, and this makes programs debugging easer.

2.4. Built-in scope

>> import builtins
>>> dir(builtins)
[‘ArithmeticError’, ‘AssertionError’, ‘AttributeError’, ‘BaseException’, ‘BufferError’, ‘BytesWarning’, ‘DeprecationWarning’, ‘EOFError’, ‘Ellipsis’,
... many other names omitted ...
‘print’, ‘property’, ‘quit’, ‘range’, ‘repr’, ‘reversed’, ‘round’, ‘set’, ‘setattr’, ‘slice’, ‘sorted’, ‘staticmethod’, ‘str’, ‘sum’, ‘super’, ‘tuple’, ‘type’, ‘vars’, ‘zip’]

- According to the LEGB rule, the interpreter searches for names in this module last.
- You get all the names from this list at your disposal by default, that is, to use them, you do not need to import any modules.
- Inside the function, you can override the variable of the built-in and other scopes:

```
def hider():
    open = ‘spam’ # Local variable, hides built-in here
    ...
    open(‘data.txt’) # Error: this no longer opens a file in this scope!
```

2.5. Global statement

The global and nonlocal instructions are the only Python instructions that resemble declaration instructions. However, they do not declare a type or size — they declare namespaces.

- Global names are names that are defined at the top level of the module.
- Global names should be declared **only if they are assigned values inside functions**.
- You can access global names inside functions without declaring them global.

```
X = 88 # Global variable X
def func():
        global X # Global variable X: outside the instruction def
        X = 99
func()
print(X)  #  99
—----------------------------------------------------
y, z = 1, 2
def all_global():
        global x
        x = y + z
print(x) #  3
```

- ***Minimize the number of global variables.***

```
X = 99
def func1():
        global X
        X = 88
def func2():
        global X
        X = 77
```

To understand this program code, you need to know the flow path of the entire program.

- ***Minimize the number of changes in other files.***

```
# first.py
X = 99 # This program code does not know about the existence of second.py

# second.py
import first
print(first.X)   # There is nothing wrong with calling a name in another file
first.X = 88     # But change might cause difficulties
```

Better add an access function:

```
# first.py
X = 99
def setX(new):
    global X
    X = new

  # second.py
import first
first.setX(88)
```

3. Arguments

- **Arguments are passed by automatically assigning objects to local variable names**. Function arguments—references to (possibly) shared objects sent by the caller—are just another instance of Python assignment at work. Because references are implemented as pointers, all arguments are, in effect, passed by pointer. Objects passed as arguments are never automatically copied.
- **Assigning to argument names inside a function does not affect the caller**. Argument names in the function header become new, local names when the function runs, in the scope of the function. There is no aliasing between function argument names and variable names in the scope of the caller.
- **Changing a mutable object argument in a function may impact the caller**. On the other hand, as arguments are simply assigned to passed-in objects, functions can change passed-in mutable objects in place, and the results may affect the caller. Mutable arguments can be input and output for functions.

3. Arguments

- **Immutable arguments are effectively passed "by value."** Objects such as integers and strings are passed by object reference instead of by copying, but because you can't change immutable objects in place anyhow, the effect is much like making a copy.
- **Mutable arguments are effectively passed "by pointer."** Objects such as lists and dictionaries are also passed by object reference.

```
def changer(a, b): # all arguments are, in effect, passed by pointer
        a= 2 # an only local variable is changed
        b[0] = 'spam' # mutable object is changed
>> X = 1
>> L= [1,2]
>> changer(X, L) # Mutable and immutable objects are transferred
>> X,L  # variable  X – wasn't changed, L — was changed
(1, ['spam', 2])
```

3. 1. Named arguments

```
from math import sqrt
def quadratic(a, b, c):
        x1 = -b / (2*a)
        x2 = sqrt(b**2 - 4*a*c) / (2*a)
        return (x1 + x2), (x1 - x2)
```

When we call this function, we can pass each of our three arguments in two different ways.

We can pass our arguments as positional arguments like this:

```
>>> quadratic(31, 93, 62)
(-1.0, -2.0)
```

Or we can pass our arguments as keyword (named) arguments like this:

```
>>> quadratic(a=31, b=93, c=62)
(-1.0, -2.0)
```

3. 1. Named arguments

The order of these arguments matters when they're passed positionally:

```
>>> quadratic(31,  93,  62)
(-1.0, -2.0)

>>> quadratic(62,  93,  31)
(-0.5, -1.0)
```

But it doesn't matter when they're passed by their name:

```
>>> quadratic(a=31,  b=93,  c=62)
(-1.0, -2.0)

>>> quadratic(c=62,  b=93,  a=31)
(-1.0, -2.0)
```

When we use keyword/named arguments, it's the name that matters, not the position.

Functions can be called with a mix of positional and named arguments:

```
>>> quadratic(31,  93,  c=62)
(-1.0, -2.0)
```

3. 1. Named arguments

When we use keyword arguments:
1.We can often leave out arguments that have default values
2.We can rearrange arguments in a way that makes them most readable
3.We call arguments by their names to make it clearer what they represent

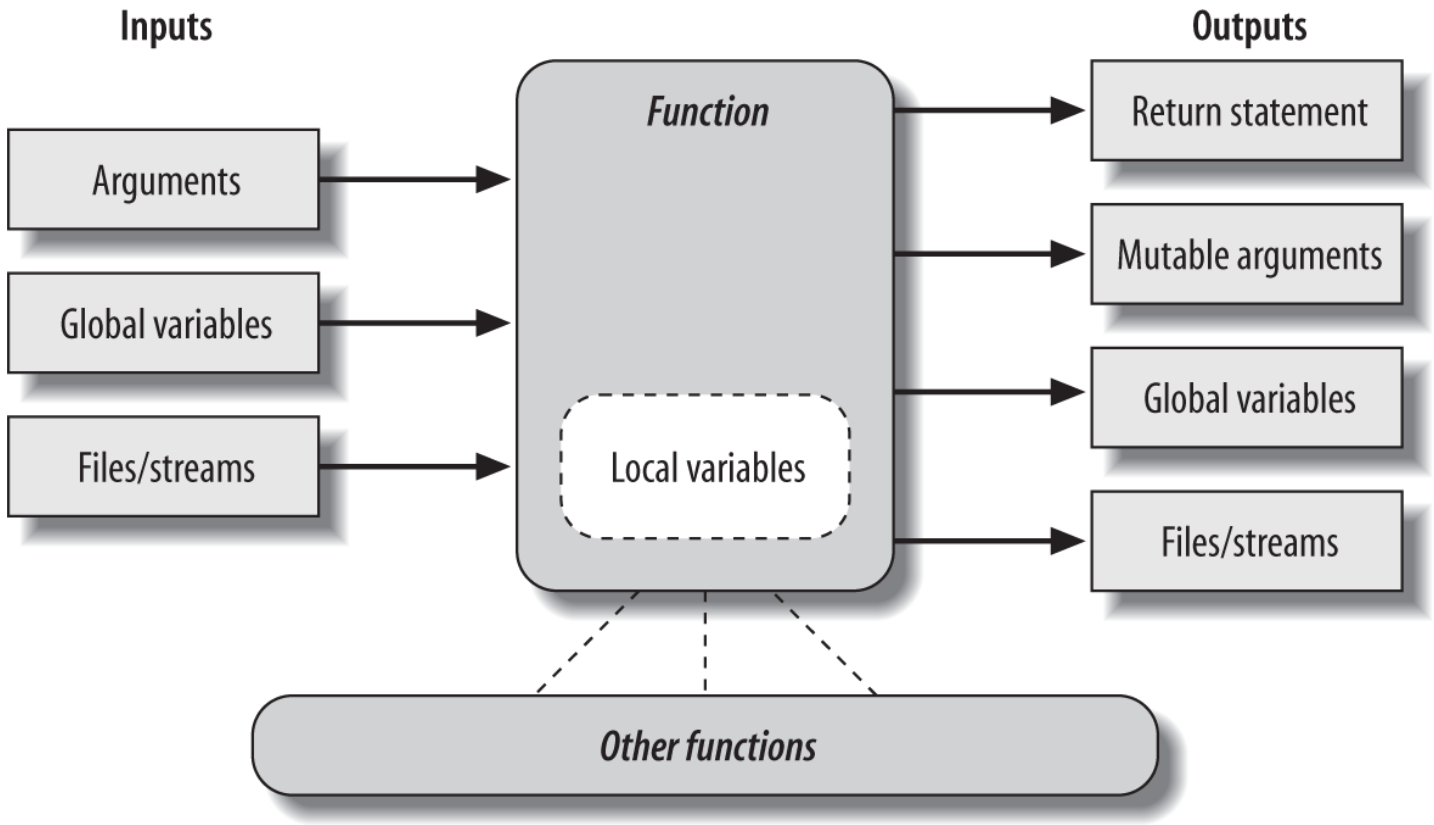
You can create a function that accepts any number of positional arguments as well as some keyword-only arguments by using the * operator to capture all the positional arguments and then specify optional keyword-only arguments:

```
def product(*numbers, initial=1):
        total = initial
        for n in numbers:
                total *= n
        return total
```

4. Function Design Concepts

- **Coupling: use arguments for inputs and return for outputs**. Generally, you should strive to make a function independent of things outside of it. Arguments and return statements are often the best ways to isolate external dependencies to a small number of well-known places in your code.
- **Coupling: use global variables only when truly necessary**. Global variables (i.e., names in the enclosing module) are usually a poor way for functions to communicate. They can create dependencies and timing issues that make programs difficult to debug, change, and reuse.
- **Coupling: don't change mutable arguments unless the caller expects it**. Functions can change parts of passed-in mutable objects, but (as with global variables) this creates a tight coupling between the caller and callee, which can make a function too specific and brittle.
- **Cohesion: each function should have a single, unified purpose**. When designed well, each of your functions should do one thing—something you can summarize in a simple declarative sentence. Otherwise, there is no way to reuse the code behind the steps mixed in the function.
- **Size: each function should be relatively small**. This naturally follows from the preceding goal, but if your functions start spanning multiple pages on your display, it's probably time to split them. Especially given that Python code is so concise to begin with, a long or deeply nested function is often a symptom of design problems. Keep it simple and keep it short.
- **Coupling: avoid changing variables in another module file directly**.

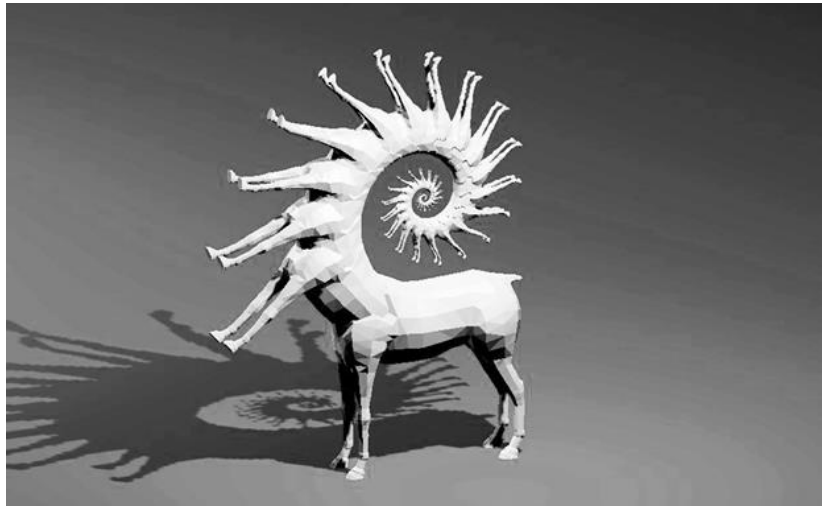
Function execution environment



5. Recursive Functions

- **Recursive functions - functions that can call themselves, directly or indirectly, forming a cycle.**
- This is a useful technique that allows you to bypass data structures with an arbitrary and unknown organization.
- Recursion is an alternative to simple loops and iterations, although not necessarily simpler or more productive.

```
def mysum(L):
        print(L)  # Help track recursion levels
            if not L: # At each level, the list L will be
shorter
                return 0
            else:
              return L[0] + mysum(L[1:]) # Calls itself
>> mysum([1, 2, 3, 4, 5])
[1,2, 3, 4, 5]
[2, 3, 4, 5]
[3, 4, 5]
[4, 5]
[5]
[ ]
15
```



*The recursive centaur. From "*The Recursive Book of Recursion"
https://inventwithpython.com/recursion/chapter1.html

5. Recursive Functions

```
def mysum(L):
        return 0 if not L else L[0] + mysum(L[1:]) # Triple operator
def mysum(L):      # summarize any types
        # assumes the presence of at least one value
        return L[0] if len(L) == 1 else L[0] + mysum(L[1:])

def mysum(L):      # Use extended
        first, *rest = L       # sequence assignment operation˘
        return first if not rest else first + mysum(rest)
>>> mysum([1]) # mysum([]) will fail in the last 2 functions 1
>>> mysum([1, 2, 3, 4, 5])
15
>>> mysum((‘s’, ‘p’, ‘a’, ‘m’)) # But they can summarize any type of data
‘spam’
>>> mysum([‘spam’, ‘ham’, ‘eggs’])
‘spamhameggs’
```

5.1. Using recursion

Consider the task of calculating the sum of all numbers in a structure consisting of nested lists:
[1, [2, [3, 4], 5], 6, [7, 8]] # Randomly nested lists
Simple loop instructions are not suitable in this case, because it will not be possible to traverse such a structure using linear iterations. Nested loop instructions also cannot be used, because lists can have unknown levels of nesting and have unknown organization. Let's traverse nested lists using recursion:

```
def sumtree(L):
        tot = 0
        for x in L: # Traversing elements of the same level
        if not isinstance(x, list):
                tot += x # numbers summarizing
        else:
                tot += sumtree(x) # Lists are handled by recursive calls
        return tot
L = [1, [2, [3, 4], 5], 6, [7, 8]] # unpredicted structure
print(sumtree(L)) # 36
# boundary cases
print(sumtree([1, [2, [3, [4, [5]]]]])) # 15 (center of gravity on the right)
print(sumtree([[[[[1], 2], 3], 4], 5])) # 15 (center of gravity on the left)
```

6. Anonymous Functions: lambda

lambda is also commonly used to code *jump tables*, which are lists or dictionaries of actions to be performed on demand. For example:

```
L = [lambda x: x**2,       # Built-in Function Definitions
     lambda x: x**3,
     lambda x: x**4]       # list from 3 function
for f in L:
    print(f(2))      # 4, 8, 16
print(L[0](3))       # 9
```

With the aid of *def:*

```
def  f1(x): return x ** 2
def  f2(x): return x ** 3     # Named Function Definitions
def  f3(x): return x ** 4
L = [f1, f2, f3]    # mutable arguments
for f in L: print(f(2))  # 4, 8, 16
print(L[0](3))  # 9
```