Functions are objects

Main concepts behind Python functions

- **def is executable code**. Python functions are written with a new statement, the **def**. Unlike functions in compiled languages such as C, **def** is an executable statement—your function does not exist until Python reaches and runs the **def**. In fact, it's legal (and even occasionally useful) to nest **def** statements inside *if* statements, *while* loops, and even other **defs**. In typical operation, **def** statements are coded in module files and are naturally run to generate functions when the module file they reside in is first imported.

Main concepts behind Python functions

- **def creates an object and assigns it to a name**. When Python reaches and runs a **def** statement, it generates a new function object and assigns it to the function's name. As with all assignments, the function name becomes a reference to the function object. There's nothing magic about the name of a function— the function object can be assigned to other names, stored in a list, and so on. Function objects may also have arbitrary user-defined attributes attached to them to record data.

Main concepts behind Python functions

- **lambda** creates an object but returns it as a result. Functions may also be created with the lambda expression, a feature that allows us to in-line function definitions in places where a def statement won't work syntactically.

Main concepts behind Python functions

- **return sends a result object back to the caller**. When a function is called, the caller stops until the function finishes its work and returns control to the caller. Functions that compute a value send it back to the caller with a return statement; the returned value becomes the result of the function call. A return without a value simply returns to the caller (and sends back None, the default result).
- **yield sends a result object back to the caller, but remembers where it left off**. Functions known as generators may also use the yield statement to send back a value and suspend their state such that they may be resumed later, to produce a series of results over time.

Main concepts behind Python functions

- **global declares module-level variables that are to be assigned**. By default, all names assigned in a function are local to that function and exist only while the function runs. To assign a name in the enclosing module, functions need to list it in a global statement. More generally, names are always looked up in scopes (namespaces) — places where variables are stored—and assignments bind names to scopes.
- **nonlocal declares enclosing function variables that are to be assigned**. Similarly, the nonlocal statement added in Python 3.X allows a function to assign a name that exists in the scope of a syntactically enclosing **def** statement. This allows enclosing functions to serve as a place to retain state—information remembered between function calls—without using shared global names.

Main concepts behind Python functions

- **Arguments are passed by assignment (object reference**). The caller and function share objects by references, but there is no name aliasing. Changing an argument name within a function does not also change the corresponding name in the caller, but changing passed-in mutable objects in place can change objects shared by the caller, and serve as a function result.
- **Arguments are passed by position, unless you say otherwise**. Values you pass in a function call match argument names in a function's definition from left to right by default. For flexibility, function calls can also pass arguments by name with name=value keyword syntax, and unpack arbitrarily many arguments to send with *pargs and **kargs starred-argument notation. Function definitions use the same two forms to specify argument defaults, and collect arbitrarily many arguments received.

Main concepts behind Python functions

- **Arguments, return values, and variables are not declared**. As with everything in Python, there are no type constraints on functions. In fact, nothing about a function needs to be declared ahead of time: you can pass in arguments of any type, return any kind of object, and so on. As one consequence, a single function can often be applied to a variety of object types—any objects that support a compatible interface (methods and expressions) will do, regardless of their specific types.
- **Immutable arguments are effectively passed "by value"** -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"** - also passed by object reference, but can be changed in place.

Polymorphism in Python

```
def times(x, y):
        return x * y
```

When Python reaches and runs this **def**, it creates a new function object that packages the function's code and assigns the object to the name **times**.

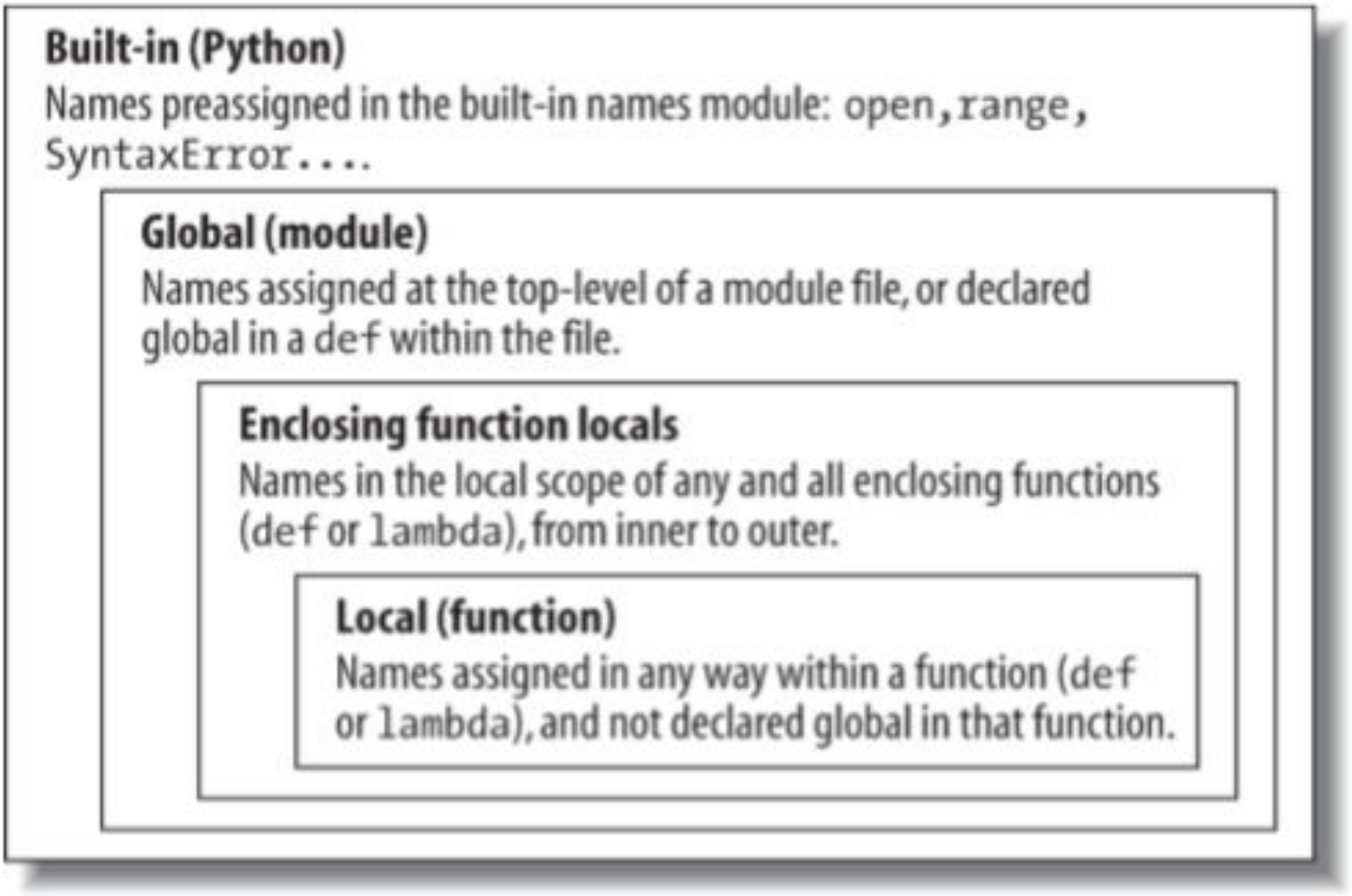
```
 times(2, 4)
times(3.14, 4)
times('Ni', 4)
```

Every operation is a polymorphic operation in Python - meaning of an operation depends on the objects being operated upon (as long as those objects support the expected interface) . This turns out to be a crucial philosophical difference between Python and statically typed languages like C++ and Java: we code to object interfaces in Python, not data types.

Scope Details

- The enclosing module is a global scope.
- The global scope spans a single file only
- Assigned names are local unless declared global or nonlocal
- All other names are enclosing function locals, globals, or built-ins
- Each call to a function creates a new local scope

Name Resolution: The LEGB Rule



LEGB: examples

```
y, z = 1, 2                    # Global variables in module
def all_global():
    global x                   # Declare globals assigned
    x = y + z                  # No need to declare y, z: LEGB rule
```

Nested Scope Examples

```
X = 99                  # Global scope name: not used
def f1():
    X = 88              # Enclosing def local
     def f2():
         print(X)       # Reference made in nested def
      f2()
f1()                    # Prints 88: enclosing def local


def f1():
  X = 88
  def f2():
     print(X)           # Remembers X in enclosing def scope
  return f2             # Return f2 but don't call it
action = f1()           # Make, return function
action()                # Call it now: prints 88
```

Factory Functions: Closures

Usually used by programs that need to generate event handlers on the fly in response to conditions at runtime (user inputs …).

```
def maker(N):
    def action(X):              # Make and return action
        return X ** N           # action retains N from enclosing scope
    return action

f = maker(2)
f(3)     # 9
f(4)     # 16
g = maker(3)
g(4)     # 64
```

We're calling the nested function that **maker** created and passed back!!! Each call to a factory function gets its own set of state information!!!

Factory Functions: Closures

```
def maker(N):
    return lambda X: X ** N        # lambda functions retain state too
h = maker(3)
 h(4)                              # 4 ** 3 again 64


def maker():
    x=4
    action = (lambda N: X ** N )   # x remembered from enclosing def
     return   action

x= maker()
 x(2)                              #  in fact  action(2):  4 ** 2 ==16
```

Factory Functions: Closures

Closures versus classes: *classes may seem better at state retention like this, because they make their memory more explicit with attribute assignments. Classes also directly support additional tools that closure functions do not, such as customization by inheritance and operator overloading, and more naturally implement multiple behaviors in the form of methods. Still, closure functions often provide a lighter-weight and viable alternative when retaining state is the only goal. They provide for per-call localized storage for data required by a single nested function.*

The nonlocal Statement in 3.X

Allows changing enclosing scope variables, as long as we declare them in nonlocal statements. With this statement, nested defs can have both read and write access to names in enclosing functions. This makes nested scope closures more useful, by providing changeable state information.

```
def tester(start):
    state = start          # Each call gets its own state
    def nested(label):
        nonlocal state     # Remembers state in enclosing scope
        print(label, state)
        state += 1         # Allowed to change it if nonlocal
    return nested
F = tester(0)              # Increments state on each call
F('spam')          #  spam 0
F('ham')           #   ham 1
F('eggs')          #   eggs 2
```

Function Objects: first-class object model

Python functions are full-blown objects, stored in pieces of memory all their own. Function objects may be assigned to other names, passed to other functions, embedded in data structures, returned from one function to another, and more, as if they were simple numbers or strings. As such, they can be freely passed around a program and called indirectly. They also support operations that have little to do with calls at all—attribute storage and annotation.

```
def echo(message):
    print(message)
echo('Direct call')
 x = echo
x('Indirect call!')
```

```
def indirect(func, arg):
        func(arg)
indirect(echo, 'Argument call!')

schedule = [ (echo, 'Spam!'), (echo,
'Ham!') ]
 for (func, arg) in schedule:
    func(arg)
```

Function Introspection

def func(a): .... *# write some function of your own*

Try it:

func.__name__

dir(func)

func.__code__

dir(func.__code__)

func.__code__.co_varnames

func.__code__.co_argcount

Function Attributes

It's possible to attach arbitrary user-defined attributes to functions. Such attributes can be used to attach state information to function objects directly, instead of using other techniques such as globals, nonlocals, and classes. Unlike nonlocals, such attributes are accessible anywhere the function itself is, even from outside its code.

Try it:
func.count = 0
func.count += 1
func.handles = 'Button-Press'
dir(func) ['__annotations__', '__call__', '__class__', '__closure__', '__code__', ...and more: in 3.X all others have double underscores so your names won't clash... __str__', '__subclasshook__', **'count', 'handles'**]

Try it

```
def f(): pass
dir(f)
len(dir(f))
[x for x in dir(f) if not x.startswith('__')]
```

You can safely use the function's namespace as though it were your own namespace or scope.

This is also a way to emulate "static locals".

Function Annotations

Annotation — arbitrary user-defined data about a function's arguments and result attached to a function object (__*annotations*__ attribute).

Compare:

```
def func(a, b, c):
    return a + b + c
func(1, 2, 3)
def func(a: 'spam', b: (1, 10), c: float) -> int:
    return a + b + c
func(1, 2, 3)
```

Python collects annotations in a dictionary and attaches it to the function object itself. Argument names become keys, the return value annotation is stored under key "return" if coded.

```
func.__annotations__
{'c': <class 'float'>, 'b': (1, 10), 'a': 'spam', 'return': <class 'int'>}
```

You can still use defaults for arguments

```
def func(a: 'spam' = 4, b: (1, 10) = 5, c: float = 6) -> int:
        return a + b + c
func(1, 2, 3)          #6
func()                 # 4 + 5 + 6   (all defaults) 15
func(1, c=10)          # 1 + 5 + 10  (keywords work normally) 16
func.__annotations__
{'c': <class 'float'>, 'b': (1, 10), 'a': 'spam', 'return': <class 'int'>}
```

Annotations are a new feature in 3.X, and some of their potential uses remain to be uncovered.

Annotations is an alternative to *function decorator arguments*.

lambdas

Lambda is also commonly used to code jump tables, which are lists or dictionaries of actions to be performed on demand.

```
L = [lambda x: x ** 2, lambda x: x ** 3,     lambda x: x ** 4]
for f in L:
    print(f(2))
print(L[0](3))

key = 'got'
{'already': (lambda: 2 + 2),     'got':     (lambda: 2 * 4),     'one':
(lambda: 2 ** 6)}[key]()         # 8

((lambda x: (lambda y: x + y))(99))(4)  #  103
```

map, filter, reduce: try it

```
counters = [1, 2, 3, 4]
list(map((lambda x: x + 3), counters))

pow(3, 4)          # 3**4

list(map(pow, [1, 2, 3], [2, 3, 4]))  # 1**2, 2**3, 3**4

list(filter((lambda x: x > 0), range(−5, 5)))

from functools import reduce
reduce((lambda x, y: x + y), [1, 2, 3, 4])    # 10
reduce((lambda x, y: x * y), [1, 2, 3, 4])    # 24

import operator, functools
functools.reduce(operator.add, [2, 4, 6])    # 12
```

Comprehensions

Apply an arbitrary expression to items in any iterable object, rather than applying a function (more general).

Compare and try:

```
res = list(map(ord, 'spam'))   # Apply function to sequence (or other)
res = [ord(x) for x in 'spam'] # Apply expression to sequence (or other)

list(map((lambda x: x ** 2), range(10)))
[x ** 2 for x in range(10)]

list(filter((lambda x: x % 2 == 0), range(5)))
[x for x in range(5) if x % 2 == 0]

list( map((lambda x: x**2), filter((lambda x: x % 2 == 0), range(10))) )
[x ** 2 for x in range(10) if x % 2 == 0]
```

Comprehensions

M = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

[M[i][i] for i in range(len(M))] *# Diagonals*

[M[i][len(M)-1-i] for i in range(len(M))]

Rewrite with loops and see the difference:

[col + 10 for row in M for col in row] *# Assign to M to retain new value*

 # [11, 12, 13, 14, 15, 16, 17, 18, 19]

[[col + 10 for col in row] for row in M] *#[[11, 12, 13], [14, 15, 16], [17, 18, 19]]*

Don't Abuse List Comprehensions: KISS!!!

On the other hand: performance, conciseness, expressiveness!!!

Generator Functions and Expressions

Generator functions are coded as normal def statements, but use **yield** statements to return results one at a time, suspending and resuming their state between each (send back a value and later be resumed, picking up where they left off).

The state that generator functions retain when they are suspended includes both their code location, and their entire local scope. Hence, their local variables retain information between results, and make it available when the functions are resumed.

Generator expressions are similar to the list comprehensions of the prior section, but they return an object that produces results on demand instead of building a result list.

Because neither constructs a result list all at once, they save memory space and allow computation time to be split across result requests. (fend on your own).

list(map(lambda x: x * 2, (1, 2, 3, 4))) *# [2, 4, 6, 8]*

list(x * 2 for x in (1, 2, 3, 4)) *# Simpler as generator? [2, 4, 6, 8]*

EIBTI: Explicit Is Better Than Implicit

Iteration protocol

Iterator objects define a **__next__** method, which either returns the next item in the iteration, or raises the special *StopIteration* exception to end the iteration. An iterable object's iterator is fetched initially with the **iter** built-in function (fend on your own).

Modules Are Objects

Modules as namespaces expose most of their interesting properties as built-in attributes so it's easy to write programs that manage other programs. We usually call such manager programs metaprograms because they work on top of other systems. This is also referred to as **introspection**, because programs can see and process object internals. It can be useful for building programming tools.

To get to an attribute called *name* in a module called *M*, we can use attribute qualification or index the module's attribute dictionary, exposed in the built-in __*dict*__ attribute. Python also exports the list of all loaded modules as the sys.modules dictionary and provides a built-in called getattr that lets us fetch attributes from their string names—it's like saying object.attr, but attr is an expression that yields a string at runtime. Because of that, all the following expressions reach the same attribute and object:

M.name

M.__dict__['name']

sys.modules['M'].name

getattr(M, 'name')

Customized version of the built-in dir function

```
#!python
""" mydir.py: a module that lists the namespaces of other modules """ seplen = 60
sepchr = '-'
def listing(module, verbose=True):
        sepline = sepchr * seplen
        if verbose:
            print(sepline)
            print('name:', module.__name__, 'file:', module.__file__)
            print(sepline)

        count = 0
        for attr in sorted(module.__dict__):
            print('%02d) %s' % (count, attr), end = ' ')
            if attr.startswith('__'):
                    print('<built-in name>')
             else:
                    print(getattr(module, attr))     # Same as .__dict__[attr]
             count += 1
             if verbose:
                print(sepline)
                print(module.__name__, 'has %d names' % count)
                 print(sepline)

if __name__ == '__main__':
import mydir
listing(mydir)                    # Self-test code: list myself
```

Problems to solve

- Think of several situations when closures present light-weighted alternative to classes. Implement them and explain.
- Code a function that is able to compute the minimum value from an arbitrary set of arguments and an arbitrary set of object data types. That is, the function should accept zero or more arguments, as many as you wish to pass. Moreover, the function should work for all kinds of Python object types: numbers, strings, lists, lists of dictionaries, files, and even None.
- Experiment with function attributes especially in closures (attributes are attached to functions generated by other factory functions, they also support multiple copy, per-call, and writeable state retention, much like nonlocal closures and class instance attributes). Try and check.
- (bonus) Emulate **zip** and **map** with Iteration Tools (see chapter 20 for help).