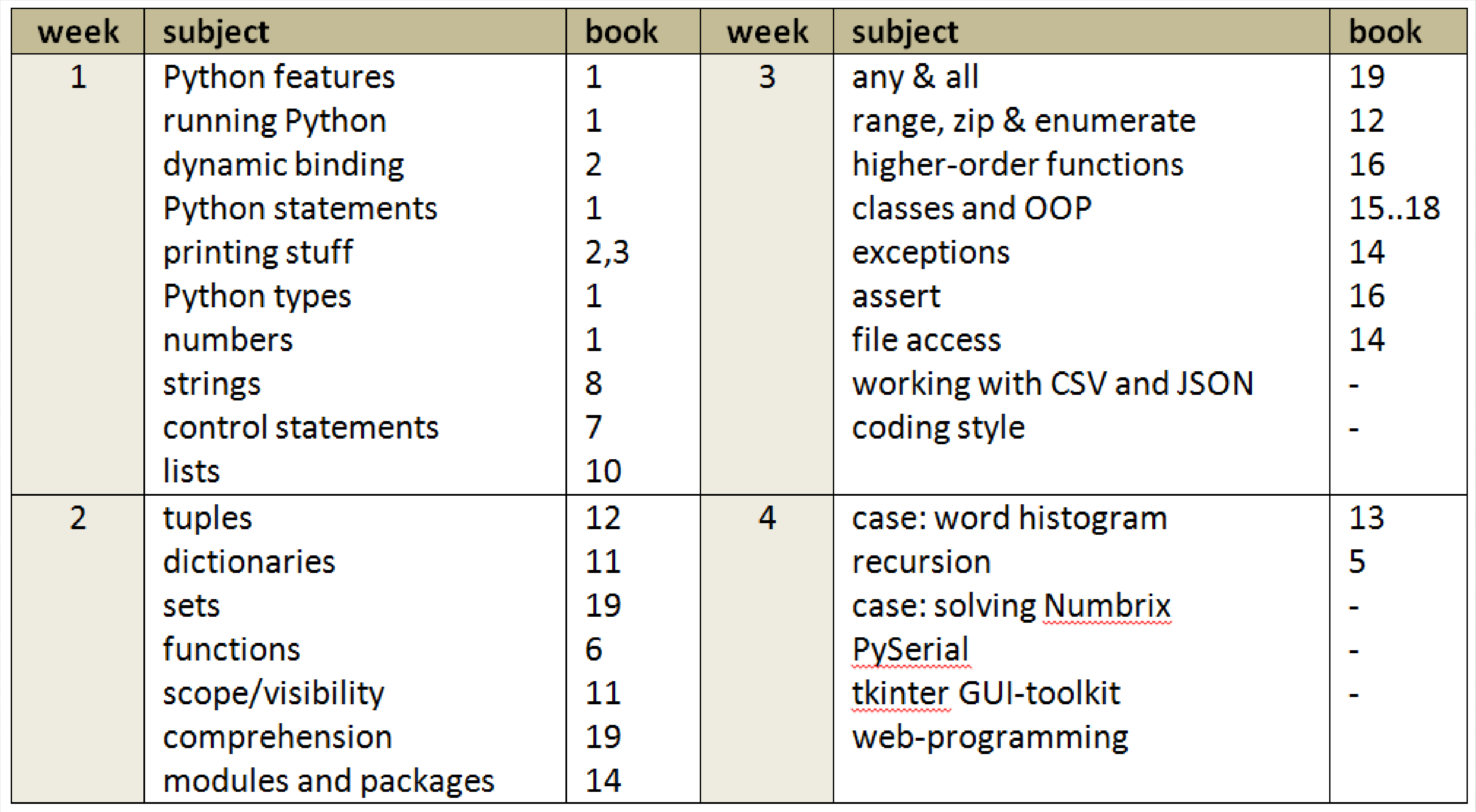
python

LECTURES WEEK 2

AGENDA



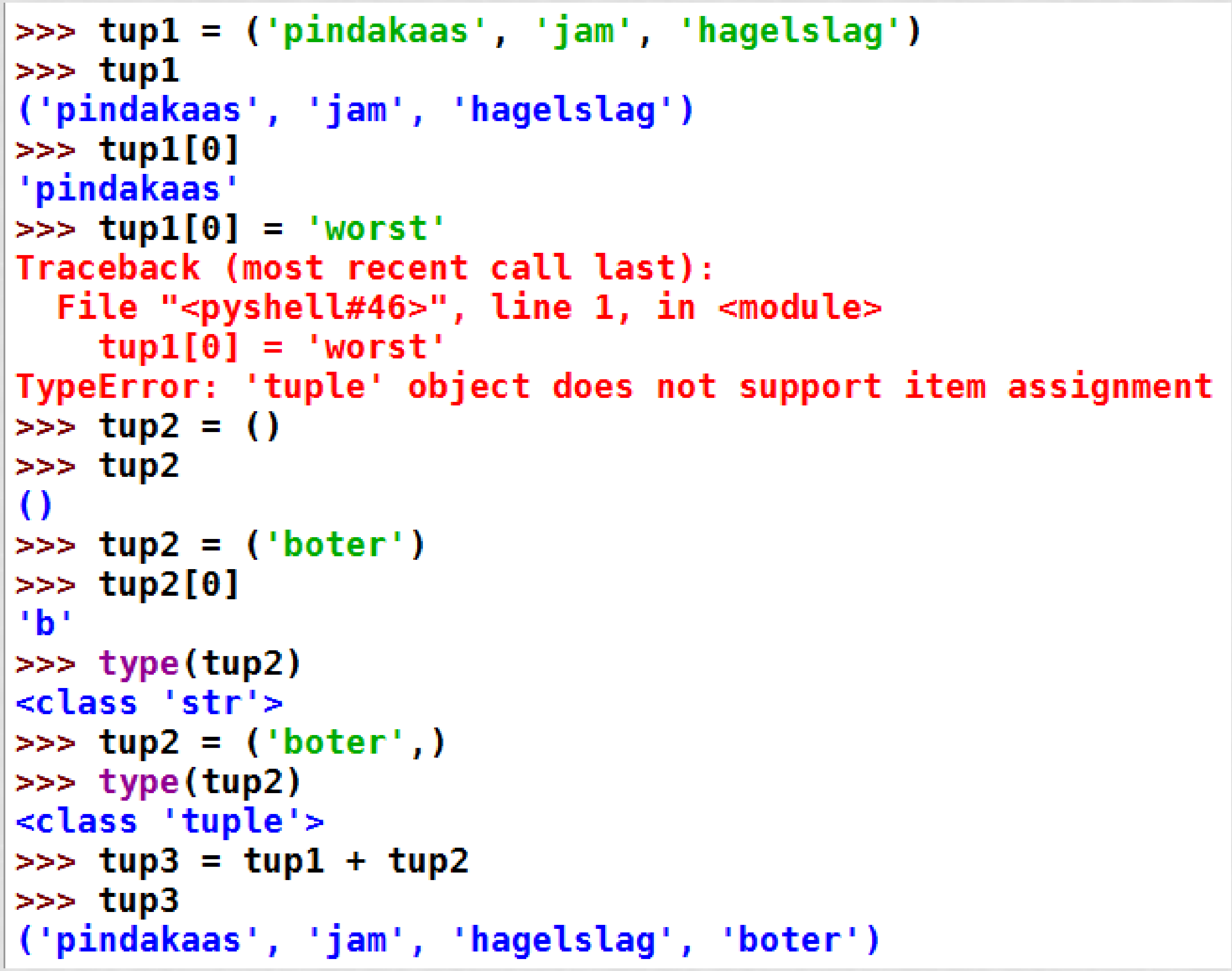
week	subject	book	week	subject	book
1	Python features	1	3	any & all	19
	running Python	1		range, zip & enumerate	12
	dynamic binding	2		higher-order functions	16
	Python statements	1		classes and OOP	15..18
	printing stuff	2,3		exceptions	14
	Python types	1		assert	16
	numbers	1		file access	14
	strings	8		working with CSV and JSON	-
	control statements	7		coding style	-
	lists	10			
2	tuples	12	4	case: word histogram	13
	dictionaries	11		recursion	5
	sets	19		case: solving Numbrix	-
	functions	6		PySerial	-
	scope/visibility	11		tkinter GUI-toolkit	-
	comprehension	19		web-programming	
	modules and packages	14			

AGENDA

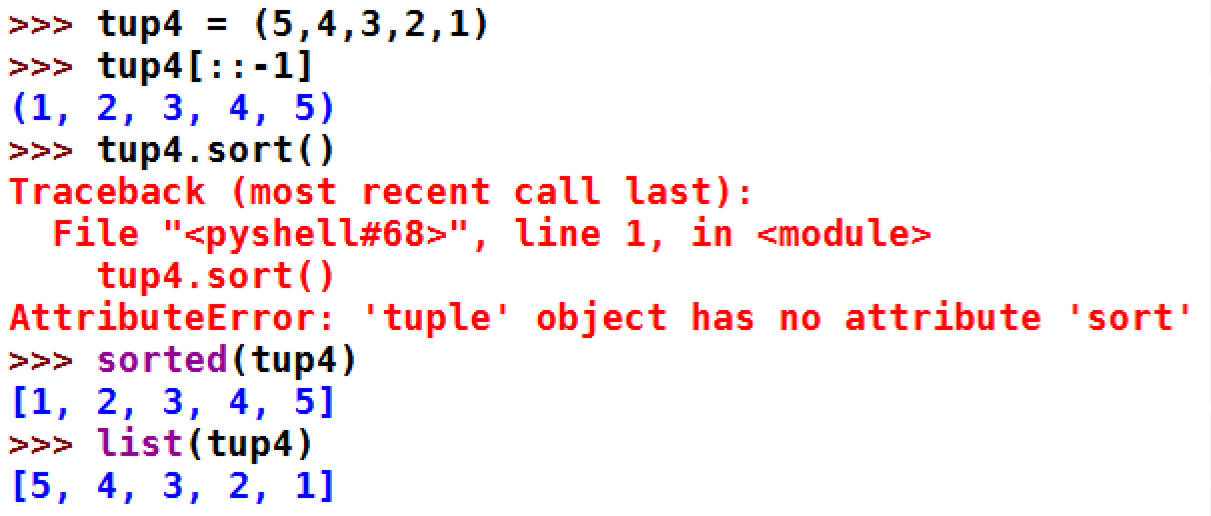
- **tuples**
- dictionaries
- sets
- functions
- scope/visibility
- comprehension
- modules and packages

TUPLES

- tuples are a like lists, but immutable
  - once a tuple is created, you cannot add, delete, replace or reorder elements
  - tuple is an (immutable) sequence
- a list uses [...], a tuple uses (...)
- can be used as :
  - immutable lists
  - records with no field names
    - `('KL 659', 'AMS', '15:30', 'JFK', '22:10')`
  - returning multiple values form a function
    - `return x, y, z`



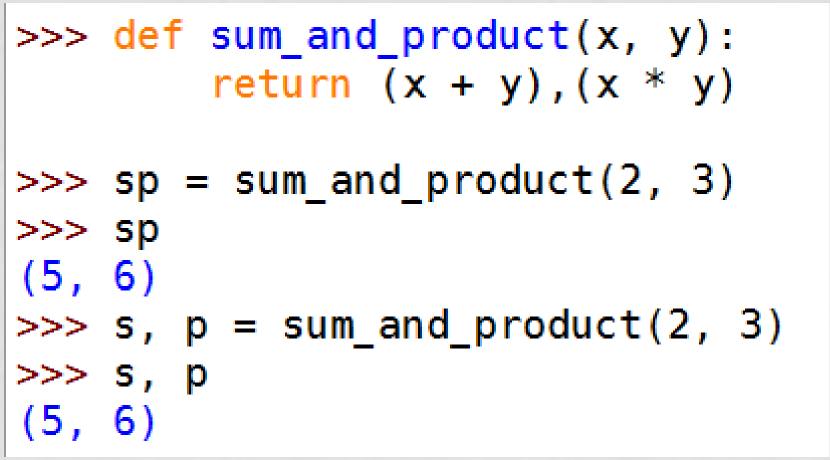
```
>>> tup1 = ('pindakaas', 'jam', 'hagelslag')
>>> tup1
('pindakaas', 'jam', 'hagelslag')
>>> tup1[0]
'pindakaas'
>>> tup1[0] = 'worst'
Traceback (most recent call last):
  File "<pyshell#46>", line 1, in <module>
    tup1[0] = 'worst'
TypeError: 'tuple' object does not support item assignment
>>> tup2 = ()
>>> tup2
()
>>> tup2 = ('boter')
>>> tup2[0]
'b'
>>> type(tup2)
<class 'str'>
>>> tup2 = ('boter',)
>>> type(tup2)
<class 'tuple'>
>>> tup3 = tup1 + tup2
>>> tup3
('pindakaas', 'jam', 'hagelslag', 'boter')
```



```
>>> tup4 = (5,4,3,2,1)
>>> tup4[::-1]
(1, 2, 3, 4, 5)
>>> tup4.sort()
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in <module>
    tup4.sort()
AttributeError: 'tuple' object has no attribute 'sort'
>>> sorted(tup4)
[1, 2, 3, 4, 5]
>>> list(tup4)
[5, 4, 3, 2, 1]
```

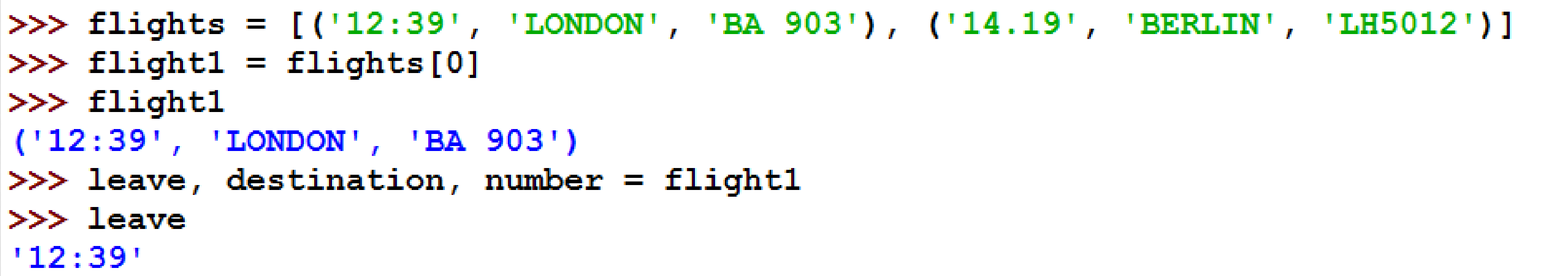
TUPLES

- tuples are a convenient way to return multiple values from functions:



```
>>> def sum_and_product(x, y):
        return (x + y),(x * y)

>>> sp = sum_and_product(2, 3)
>>> sp
(5, 6)
>>> s, p = sum_and_product(2, 3)
>>> s, p
(5, 6)
```

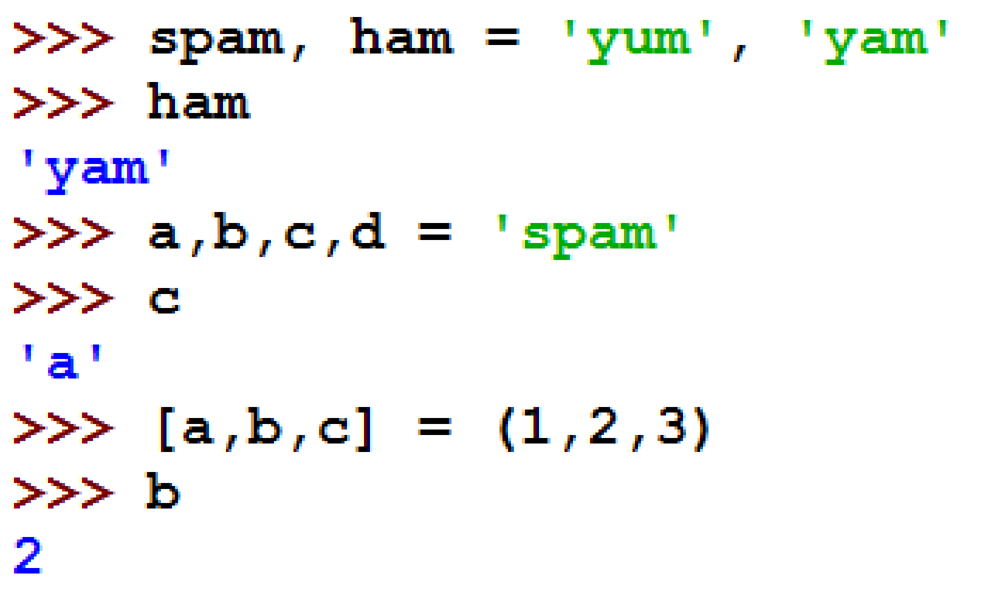
- tuples can be used for multiple assignment:



```
>>> flights = [('12:39', 'LONDON', 'BA 903'), ('14.19', 'BERLIN', 'LH5012')]
>>> flight1 = flights[0]
>>> flight1
('12:39', 'LONDON', 'BA 903')
>>> leave, destination, number = flight1
>>> leave
'12:39'
```

SEQUENCE UNPACKING

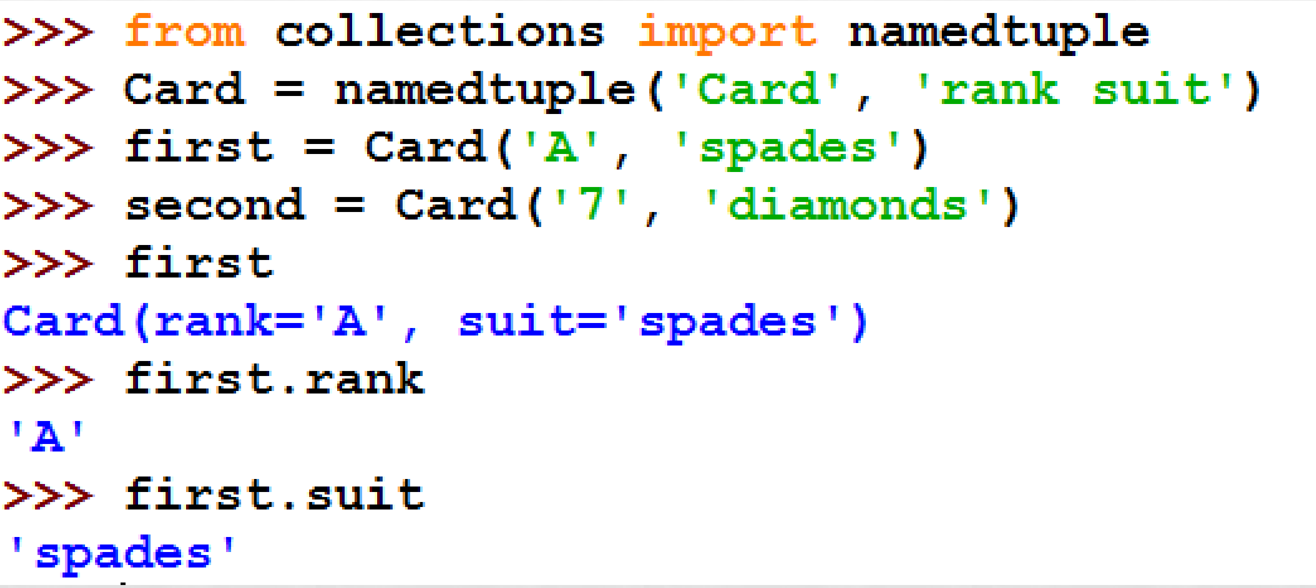
- sequence unpacking : # items left = # items right



```
>>> spam, ham = 'yum', 'yam'
>>> ham
'yam'
>>> a,b,c,d = 'spam'
>>> c
'a'
>>> [a,b,c] = (1,2,3)
>>> b
2
```

NAMED TUPLE

- allows access fields by name instead of index
- create a named tuple with a class name and a list of field names
  - a named tuple is a regular Python class
  - *name* and *reference* should be the same



```
>>> from collections import namedtuple
>>> Card = namedtuple('Card', 'rank suit')
>>> first = Card('A', 'spades')
>>> second = Card('7', 'diamonds')
>>> first
Card(rank='A', suit='spades')
>>> first.rank
'A'
>>> first.suit
'spades'
```

ASSIGNMENTS

```
a) tup = (1,2,3,4,5,6,7,8,9,10)
   output: (3,4,5)

b) L = [("Bob", 19, "CS"), ("Mary", 21, "EE"),
   ("Alice", 20, "CE")]
   output: ['CS', 'EE', 'CE']

c) replace last value of tuple
   L = [(10, 20, 40), (40, 50, 60), (70, 80, 90)]
   output: [(10, 20, 100), (40, 50, 100), (70, 80, 100)]
```

ASSIGNMENTS

d) sort a tuple by its float element

```
price = [('item1','12.20'),('item2','15.10'),
         ('item3','24.5')]
output: [('item3','24.5'),('item2','15.10'),
         ('item1', '12.20')]
```

SOLUTIONS

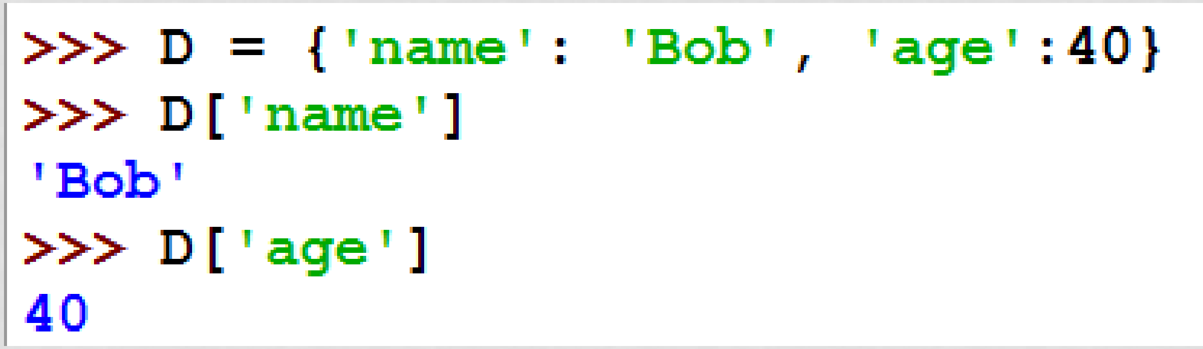
```
a) tup[2:5]
b) [e[2] for e in L]
c) [e[:-1] + (100,) for e in L]
d) sorted(price, key=lambda x: float(x[1]), reverse=True))

   or in-place:
   price.sort(key=lambda x:float(x[1]), reverse=True)
```

AGENDA

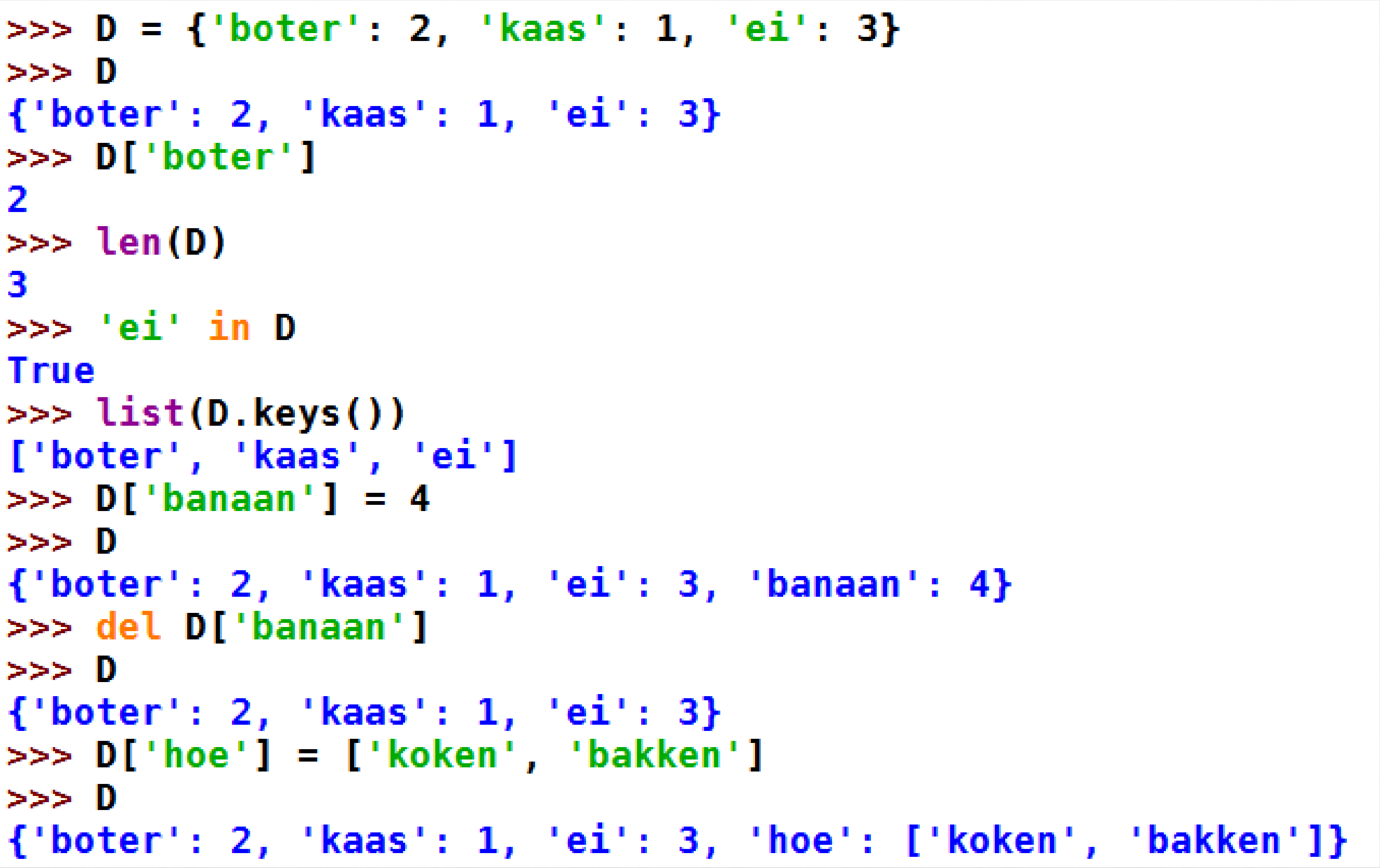
- tuples
- **dictionaries**
- sets
- functions
- scope/visibility
- comprehension
- modules and packages

DICTIONARY

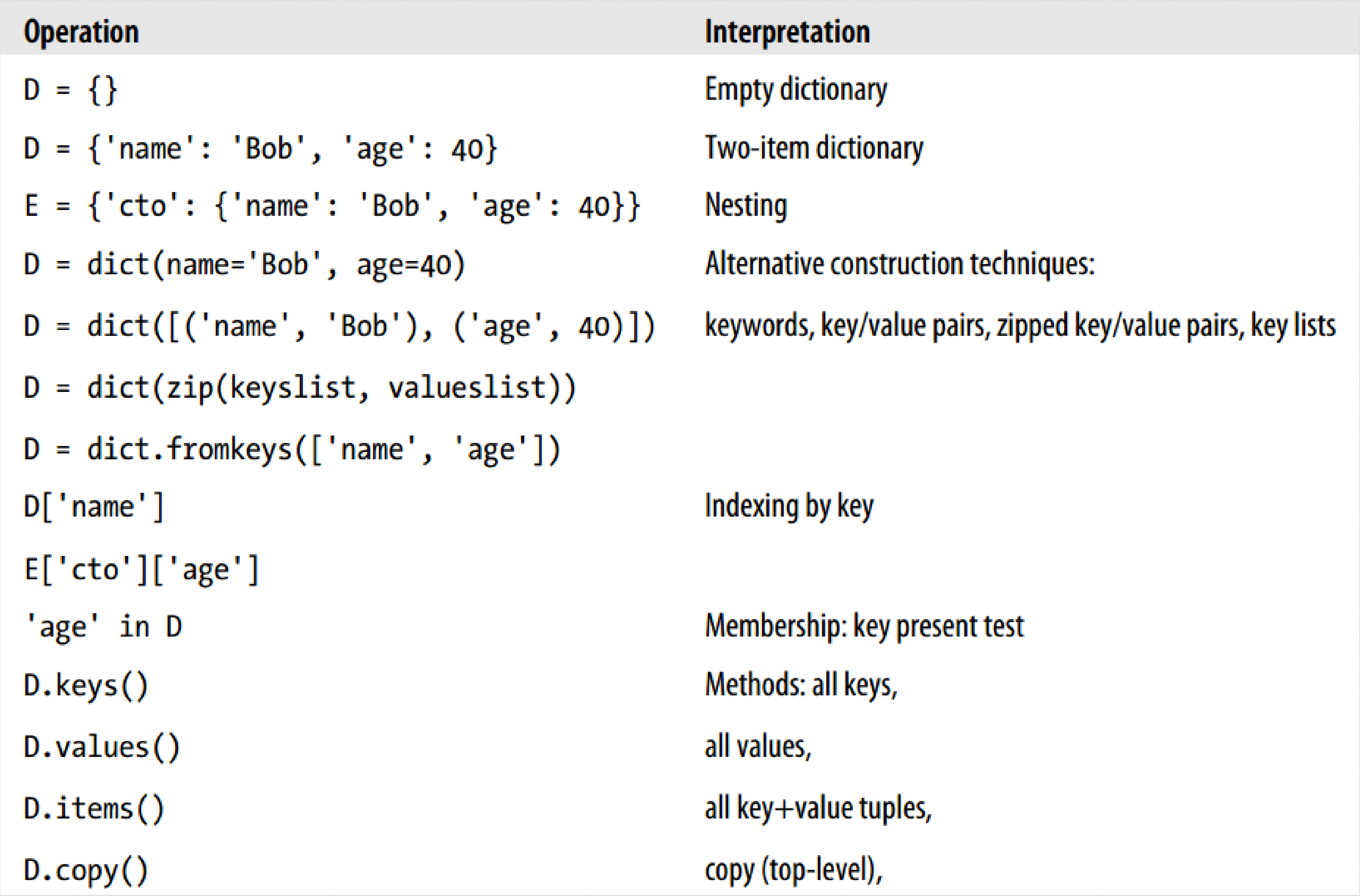
- key, value pairs
- dictionaries are mutable, but keys must be immutable (= 'hashable')
  - value of a key can never change
- unlike lists or tuples, items in dictionaries are unordered



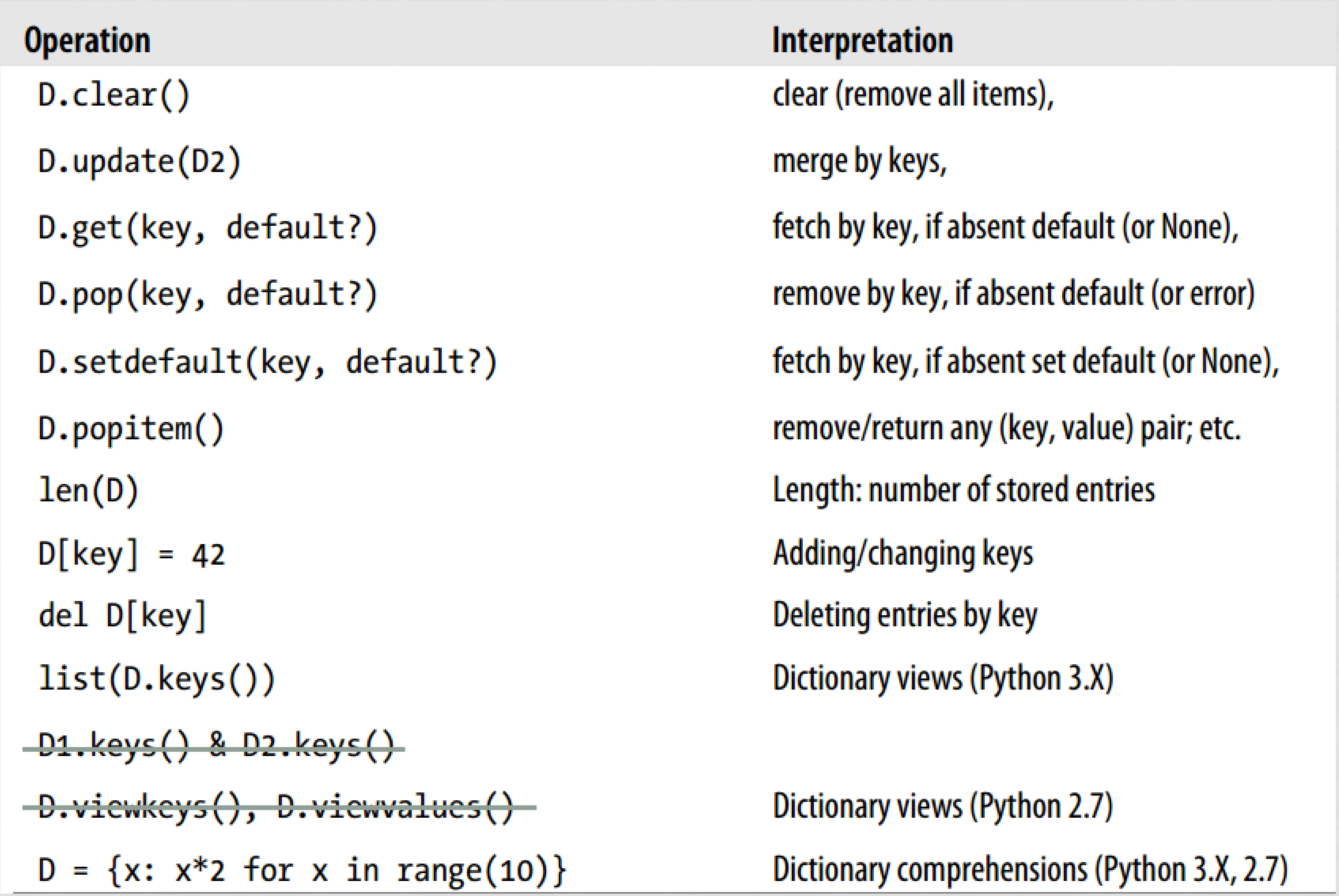
```
>>> D = {'name': 'Bob', 'age':40}
>>> D['name']
'Bob'
>>> D['age']
40
```



```
>>> D = {'boter': 2, 'kaas': 1, 'ei': 3}
>>> D
{'boter': 2, 'kaas': 1, 'ei': 3}
>>> D['boter']
2
>>> len(D)
3
>>> 'ei' in D
True
>>> list(D.keys())
['boter', 'kaas', 'ei']
>>> D['banaan'] = 4
>>> D
{'boter': 2, 'kaas': 1, 'ei': 3, 'banaan': 4}
>>> del D['banaan']
>>> D
{'boter': 2, 'kaas': 1, 'ei': 3}
>>> D['hoe'] = ['koken', 'bakken']
>>> D
{'boter': 2, 'kaas': 1, 'ei': 3, 'hoe': ['koken', 'bakken']}
```



Operation	Interpretation
`D = {}`	Empty dictionary
`D = {'name': 'Bob', 'age': 40}`	Two-item dictionary
`E = {'cto': {'name': 'Bob', 'age': 40}}`	Nesting
`D = dict(name='Bob', age=40)`	Alternative construction techniques:
`D = dict([('name', 'Bob'), ('age', 40)])`	keywords, key/value pairs, zipped key/value pairs, key lists
`D = dict(zip(keyslist, valueslist))`	
`D = dict.fromkeys(['name', 'age'])`	
`D['name']`	Indexing by key
`E['cto']['age']`	
`'age' in D`	Membership: key present test
`D.keys()`	Methods: all keys,
`D.values()`	all values,
`D.items()`	all key+value tuples,
`D.copy()`	copy (top-level),

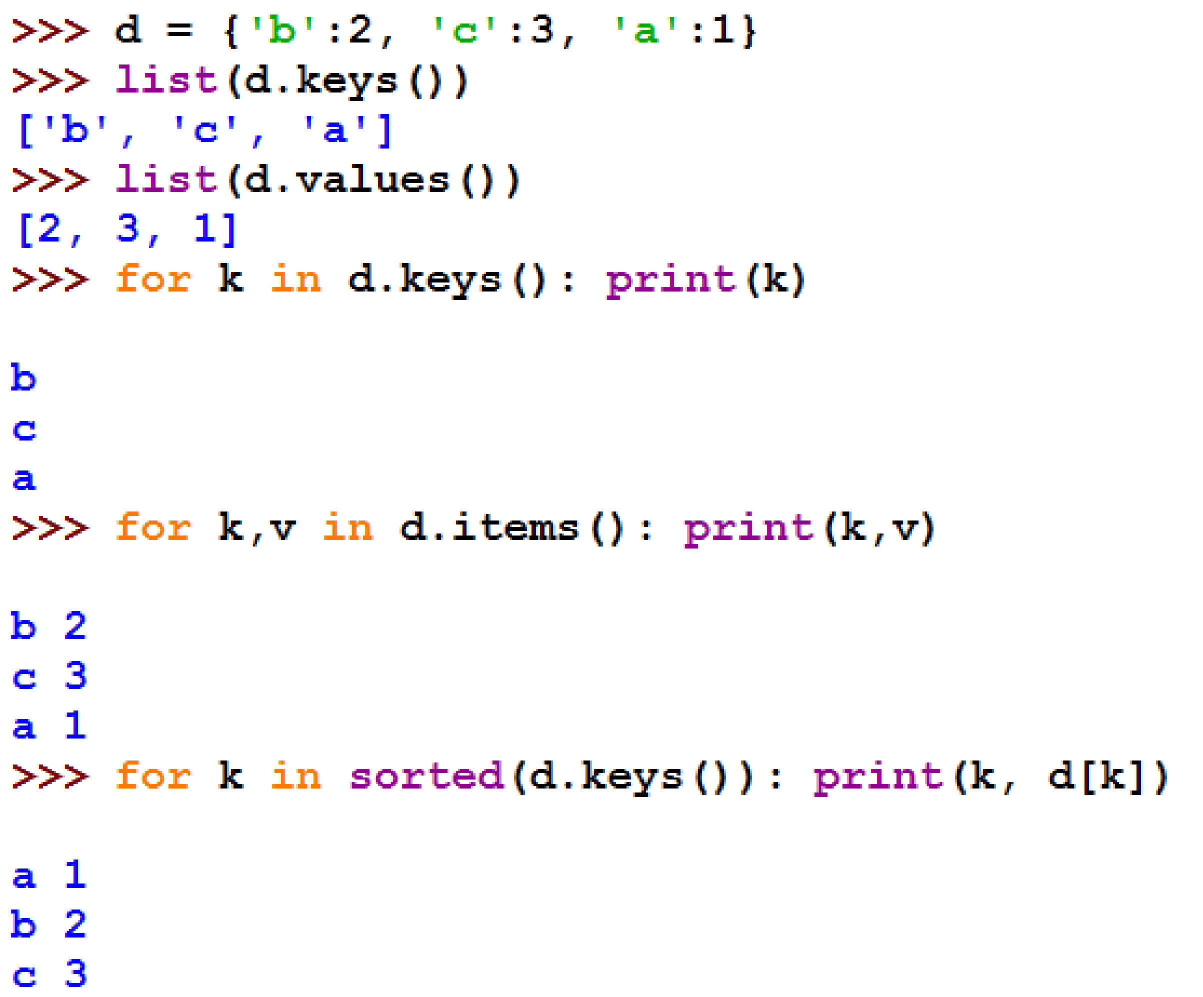


Operation	Interpretation
`D.clear()`	clear (remove all items),
`D.update(D2)`	merge by keys,
`D.get(key, default?)`	fetch by key, if absent default (or None),
`D.pop(key, default?)`	remove by key, if absent default (or error)
`D.setdefault(key, default?)`	fetch by key, if absent set default (or None),
`D.popitem()`	remove/return any (key, value) pair; etc.
`len(D)`	Length: number of stored entries
`D[key] = 42`	Adding/changing keys
`del D[key]`	Deleting entries by key
`list(D.keys())`	Dictionary views (Python 3.X)
~~`D1.keys() & D2.keys()`~~	
~~`D.viewkeys(), D.viewvalues()`~~	Dictionary views (Python 2.7)
`D = {x: x*2 for x in range(10)}`	Dictionary comprehensions (Python 3.X, 2.7)

PRINTING KEYS AND VALUES

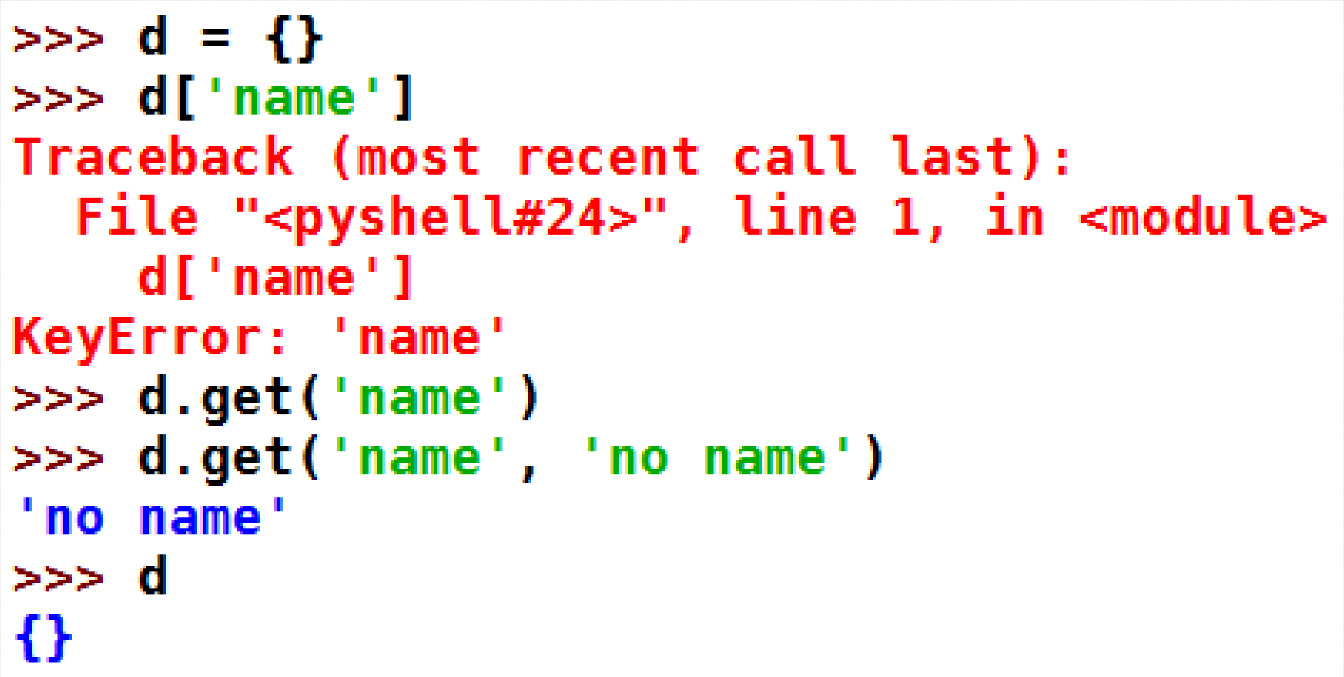
- the keys(), values(), and items() methods return an object which is generally used to iterate over in for loops
- result is not a list, but a so called view object
- you have to do `list(d.keys())` etc.

PRINTING KEYS AND VALUES



```
>>> d = {'b':2, 'c':3, 'a':1}
>>> list(d.keys())
['b', 'c', 'a']
>>> list(d.values())
[2, 3, 1]
>>> for k in d.keys(): print(k)

b
c
a
>>> for k,v in d.items(): print(k,v)

b 2
c 3
a 1
>>> for k in sorted(d.keys()): print(k, d[k])

a 1
b 2
c 3
```

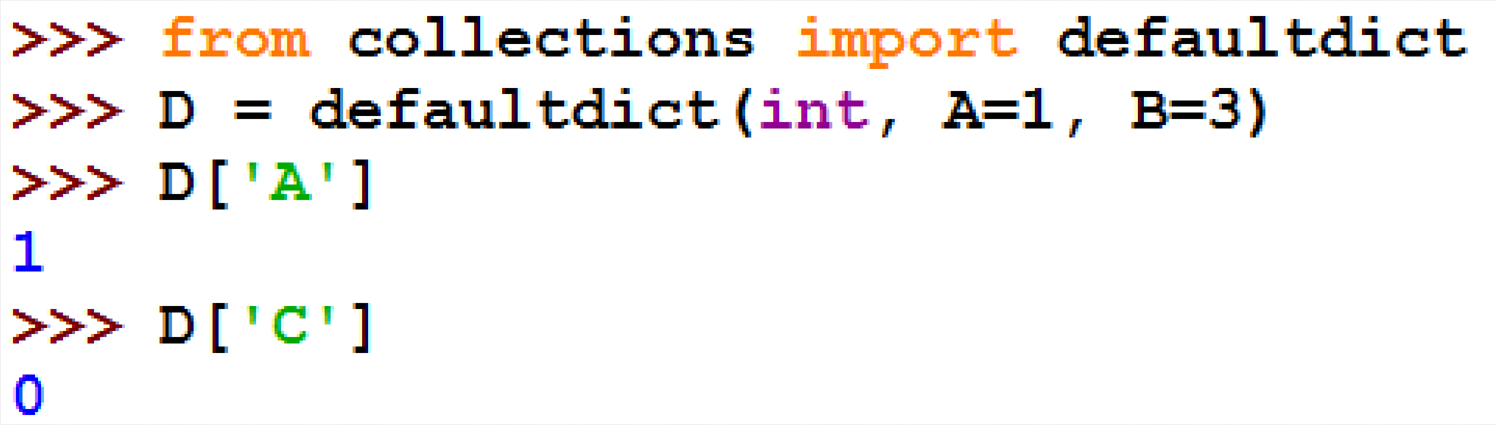
GIVE A DEFAULT VALUE



```
>>> d = {}
>>> d['name']
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    d['name']
KeyError: 'name'
>>> d.get('name')
>>> d.get('name', 'no name')
'no name'
>>> d
{}
```

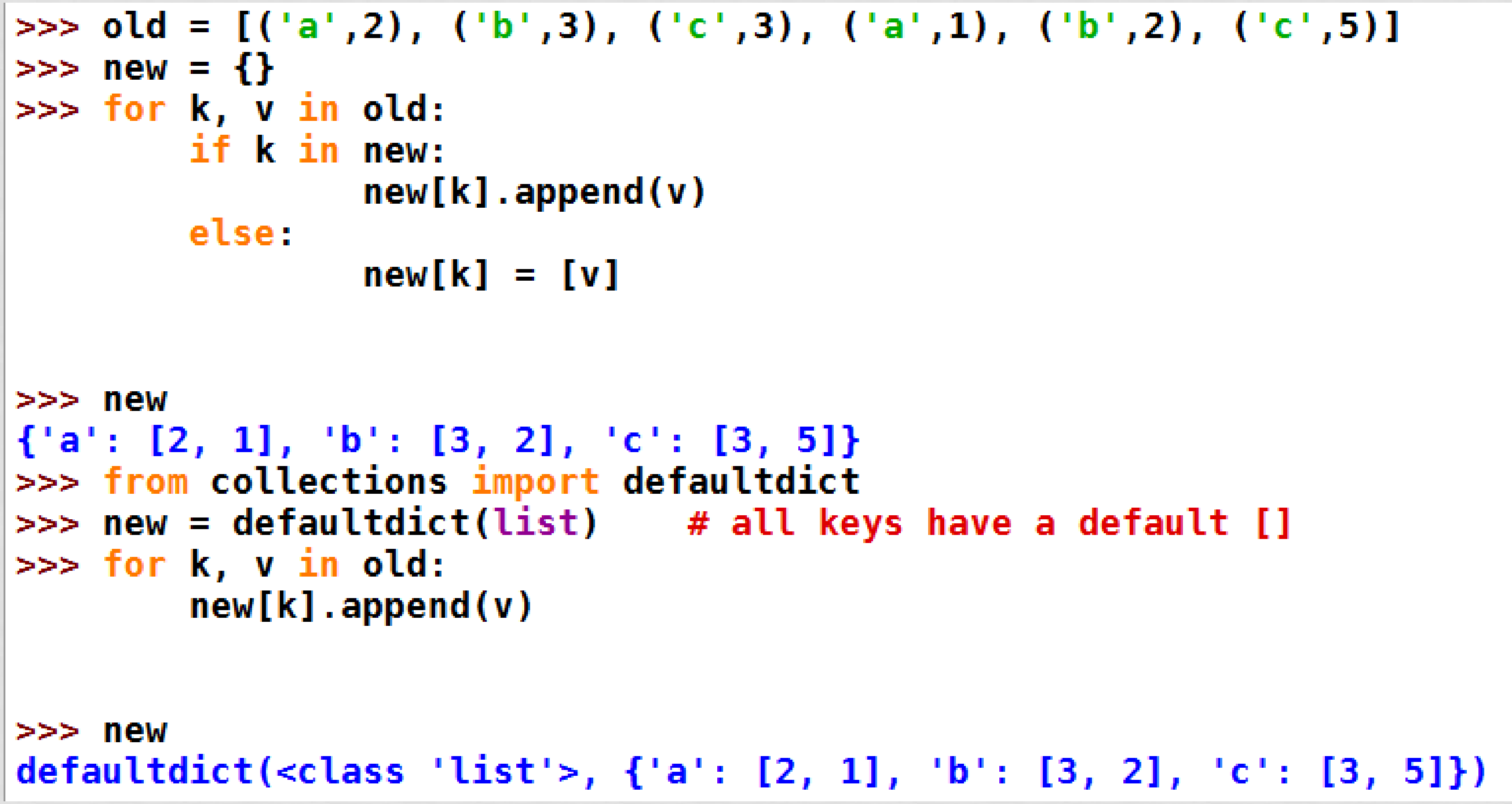
DEFAULTDICT

- if you want a default values for keys that don't exist (yet) in the dictionary
- a defaultdict will create a key with the default value when you access a key that doesn’t exists
  - using a zero-argument function, usually int or list (returning 0 and [ ])



```
>>> from collections import defaultdict
>>> D = defaultdict(int, A=1, B=3)
>>> D['A']
1
>>> D['C']
0
```

DEFAULTDICT

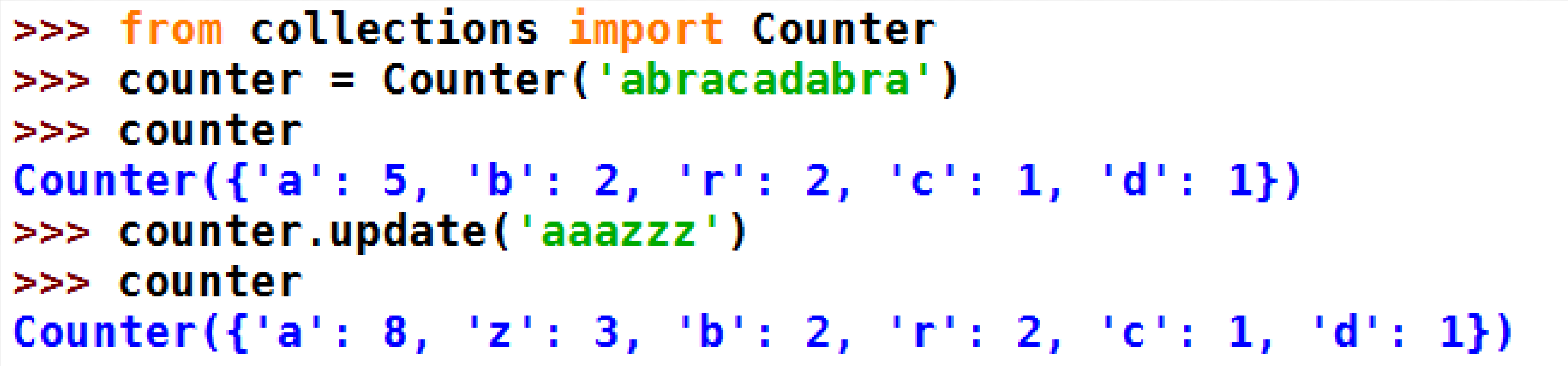


```
>>> old = [('a',2), ('b',3), ('c',3), ('a',1), ('b',2), ('c',5)]
>>> new = {}
>>> for k, v in old:
        if k in new:
                new[k].append(v)
        else:
                new[k] = [v]


>>> new
{'a': [2, 1], 'b': [3, 2], 'c': [3, 5]}
>>> from collections import defaultdict
>>> new = defaultdict(list)     # all keys have a default []
>>> for k, v in old:
        new[k].append(v)


>>> new
defaultdict(<class 'list'>, {'a': [2, 1], 'b': [3, 2], 'c': [3, 5]})
```

COLLECTIONS.COUNTER

- a collection where elements are stored as dictionary keys and their counts are stored as dictionary values
- constructor expects iterable or mapping
- most_common method returns a list of value-frequency pairs, sorted from most common to least

COLLECTIONS.COUNTER



```
>>> from collections import Counter
>>> counter = Counter('abracadabra')
>>> counter
Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
>>> counter.update('aaazzz')
>>> counter
Counter({'a': 8, 'z': 3, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
```

ASSIGNMENTS

```
a) get most expensive item
   d = {'it1':45.50, 'it2':35, 'it3':41.30, 'it4':55, 'it5': 24}
   result: 55

b) merge d1 and d2
   d1 = {'a': 100, 'b': 200}
   d2 = {'x': 300, 'y': 200}
   result: {'x': 300, 'y': 200, 'a': 100, 'b': 200}

c) add values with same key
   d1 = {'a':100, 'b':200, 'c':300}
   d2 = {'a':300, 'b':200, 'd':400}
   result: Counter({'a':400, 'b':400, 'd':400, 'c':300})
```

SOLUTIONS

```
a) max(d.values())

b) d = d1.copy() # d = d1 will not make a copy
   d.update(d2)
   d

c) from collections import Counter
   Counter(d1) + Counter(d2)
```

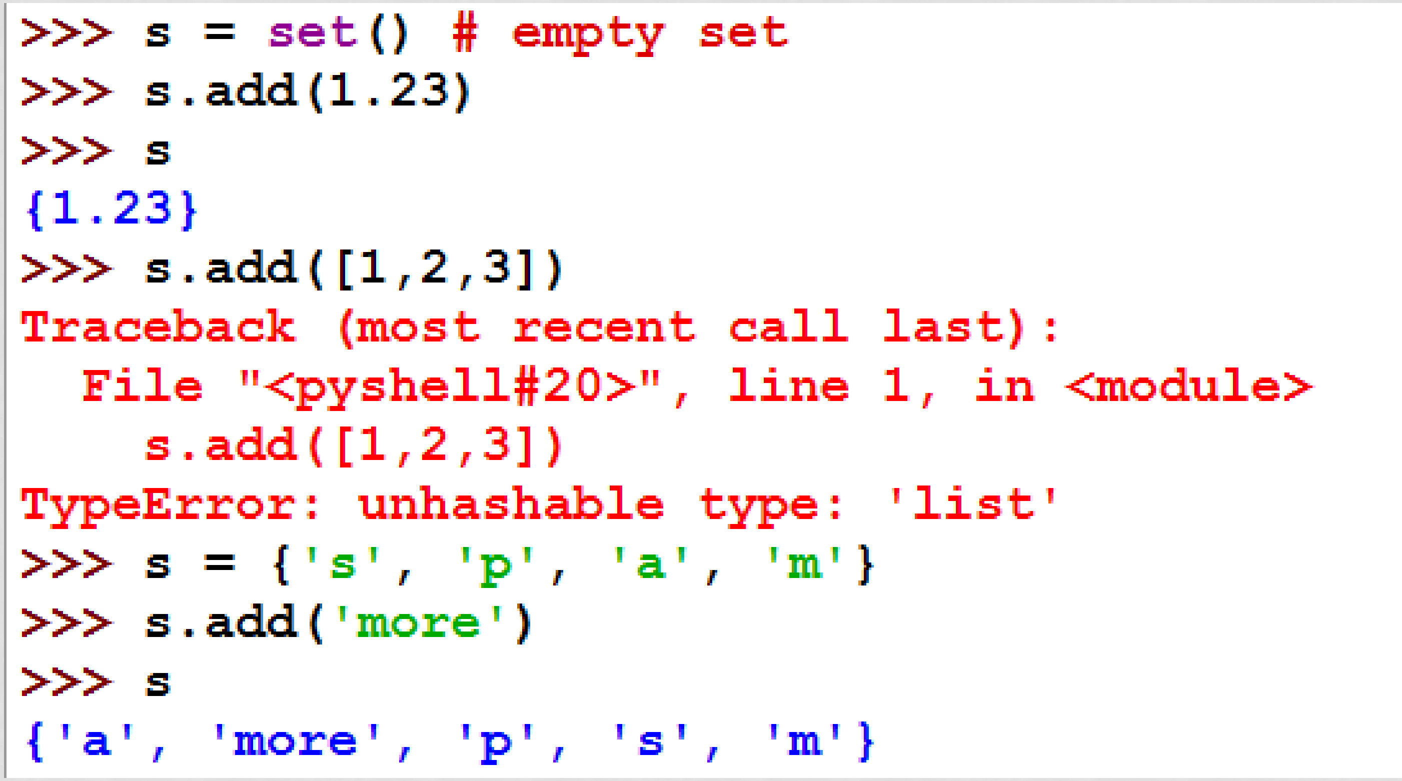
AGENDA

- tuples
- dictionaries
- **sets**
- functions
- scope/visibility
- comprehension
- modules and packages

SETS

- sets have no duplicate elements
- items in sets are unordered
  - element ordering depends on insertion order
- only immutable elements are allowed
- sets have significant memory overhead
  - because hash tables are sparse arrays (arrays that always have empty cells/buckets)

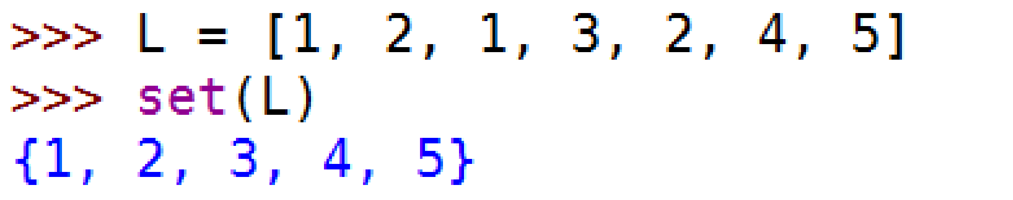
SETS



```
>>> s = set() # empty set
>>> s.add(1.23)
>>> s
{1.23}
>>> s.add([1,2,3])
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    s.add([1,2,3])
TypeError: unhashable type: 'list'
>>> s = {'s', 'p', 'a', 'm'}
>>> s.add('more')
>>> s
{'a', 'more', 'p', 's', 'm'}
```

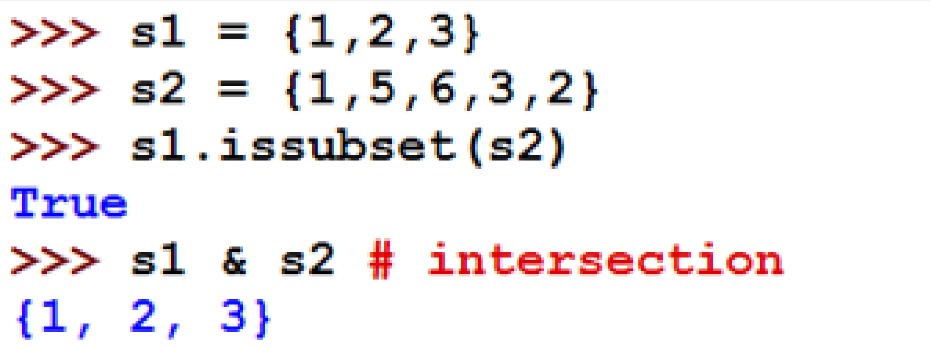
SETS

- sets can be used to filter duplicates out of other collections



```
>>> L = [1, 2, 1, 3, 2, 4, 5]
>>> set(L)
{1, 2, 3, 4, 5}
```

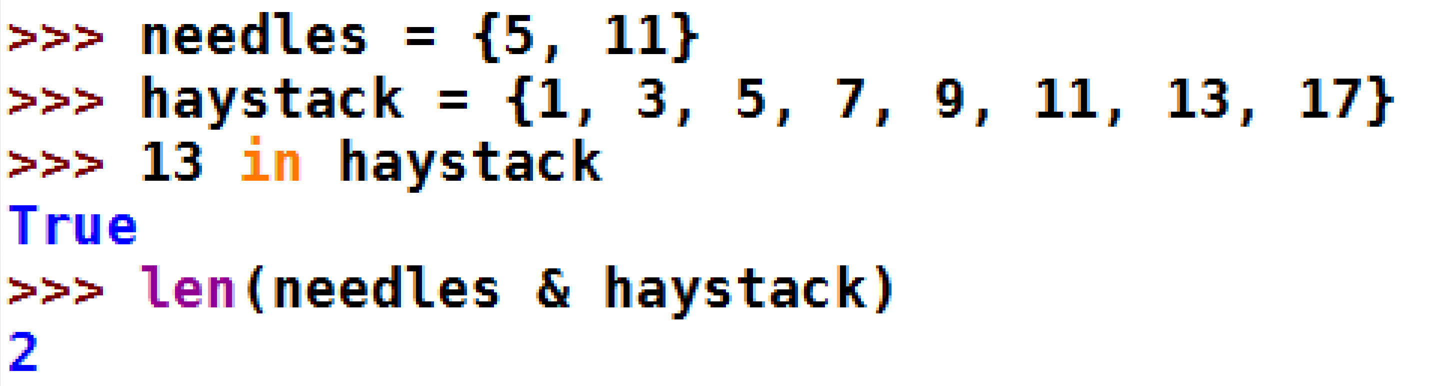
- some set operations



```
>>> s1 = {1,2,3}
>>> s2 = {1,5,6,3,2}
>>> s1.issubset(s2)
True
>>> s1 & s2 # intersection
{1, 2, 3}
```

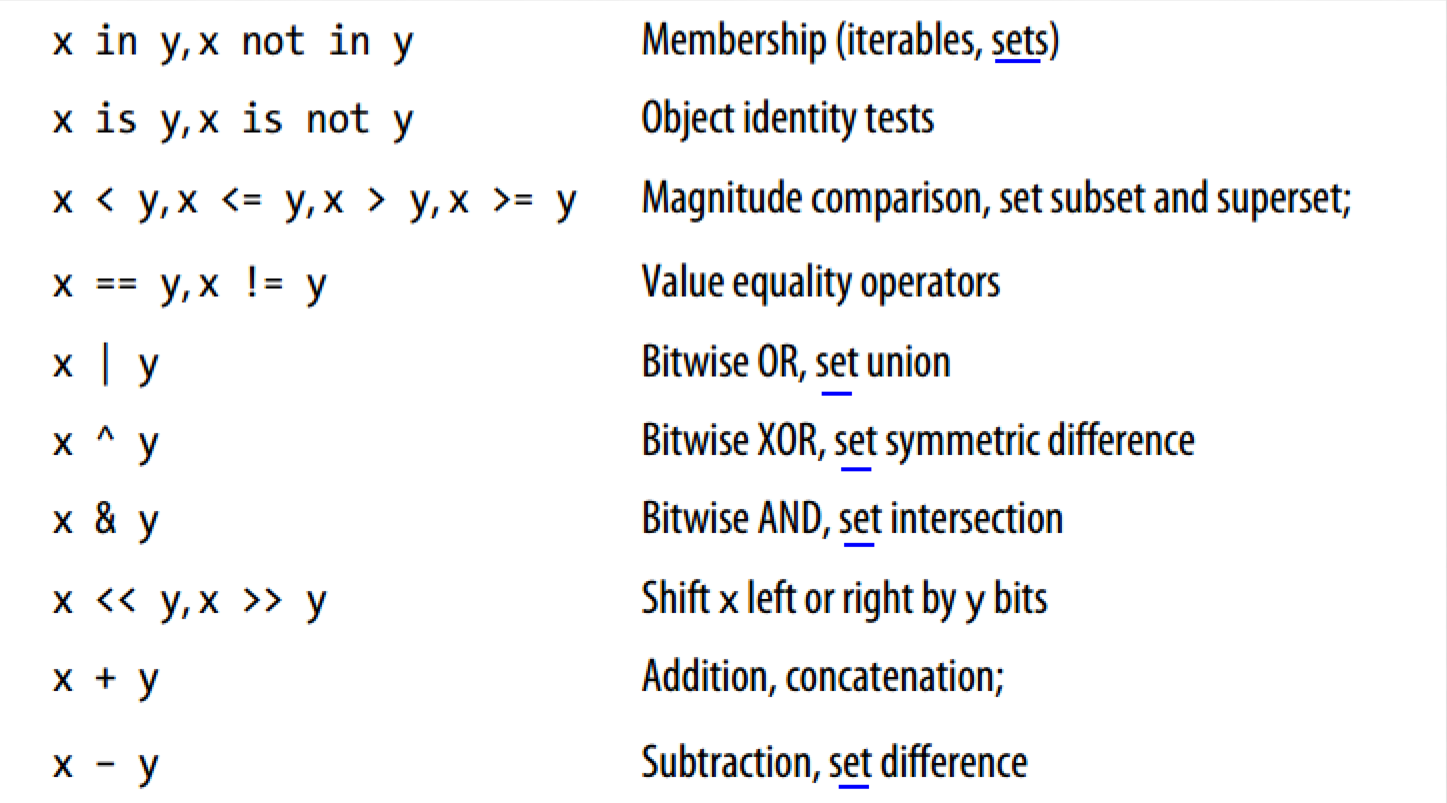
MEMBERSHIP TESTING

- membership tests are very fast, since set members are hashed



```
>>> needles = {5, 11}
>>> haystack = {1, 3, 5, 7, 9, 11, 13, 17}
>>> 13 in haystack
True
>>> len(needles & haystack)
2
```

SOME SET OPERATORS

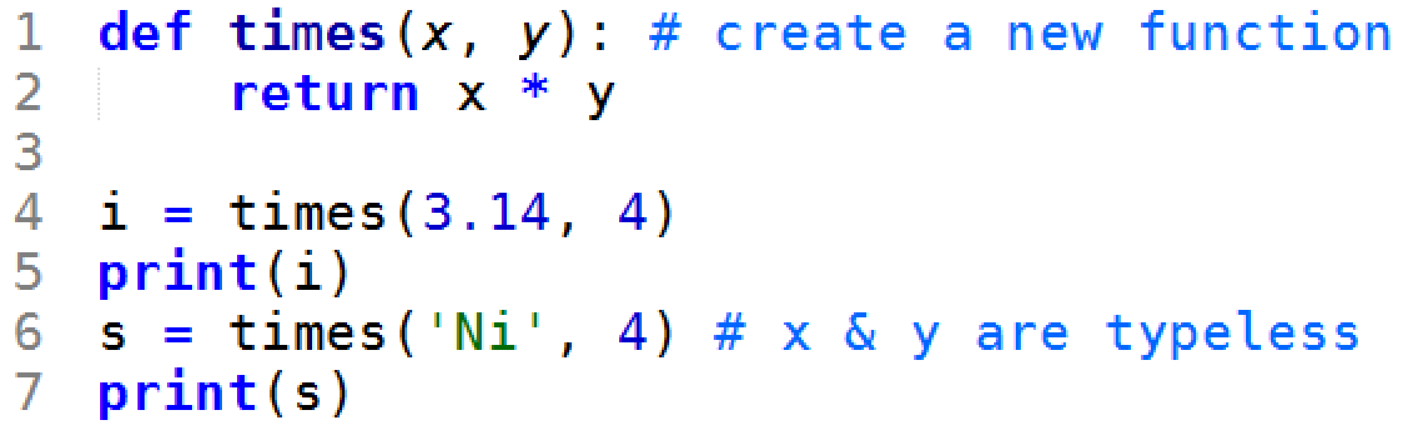


`x in y, x not in y`	Membership (iterables, sets)
`x is y, x is not y`	Object identity tests
`x < y, x <= y, x > y, x >= y`	Magnitude comparison, set subset and superset;
`x == y, x != y`	Value equality operators
`x | y`	Bitwise OR, set union
`x ^ y`	Bitwise XOR, set symmetric difference
`x & y`	Bitwise AND, set intersection
`x << y, x >> y`	Shift x left or right by y bits
`x + y`	Addition, concatenation;
`x - y`	Subtraction, set difference

AGENDA

- tuples
- dictionaries
- sets
- **functions**
- scope/visibility
- comprehension
- modules and packages

DEF



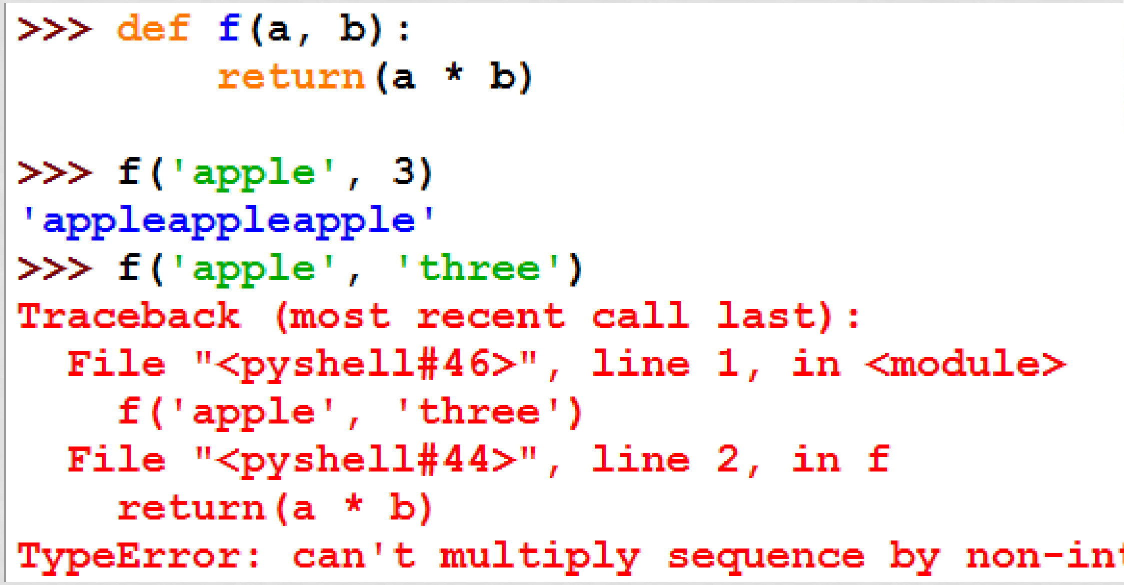
```python
def times(x, y): # create a new function
    return x * y

i = times(3.14, 4)
print(i)
s = times('Ni', 4) # x & y are typeless
print(s)
```

DEF

- def creates a new function object and assigns it to a name
- code inside a def is not evaluated until the function is actually called
- without return statement None object is returned
- in Python function arguments are type-less
  - a function may work on any combination of objects supporting the expected interface

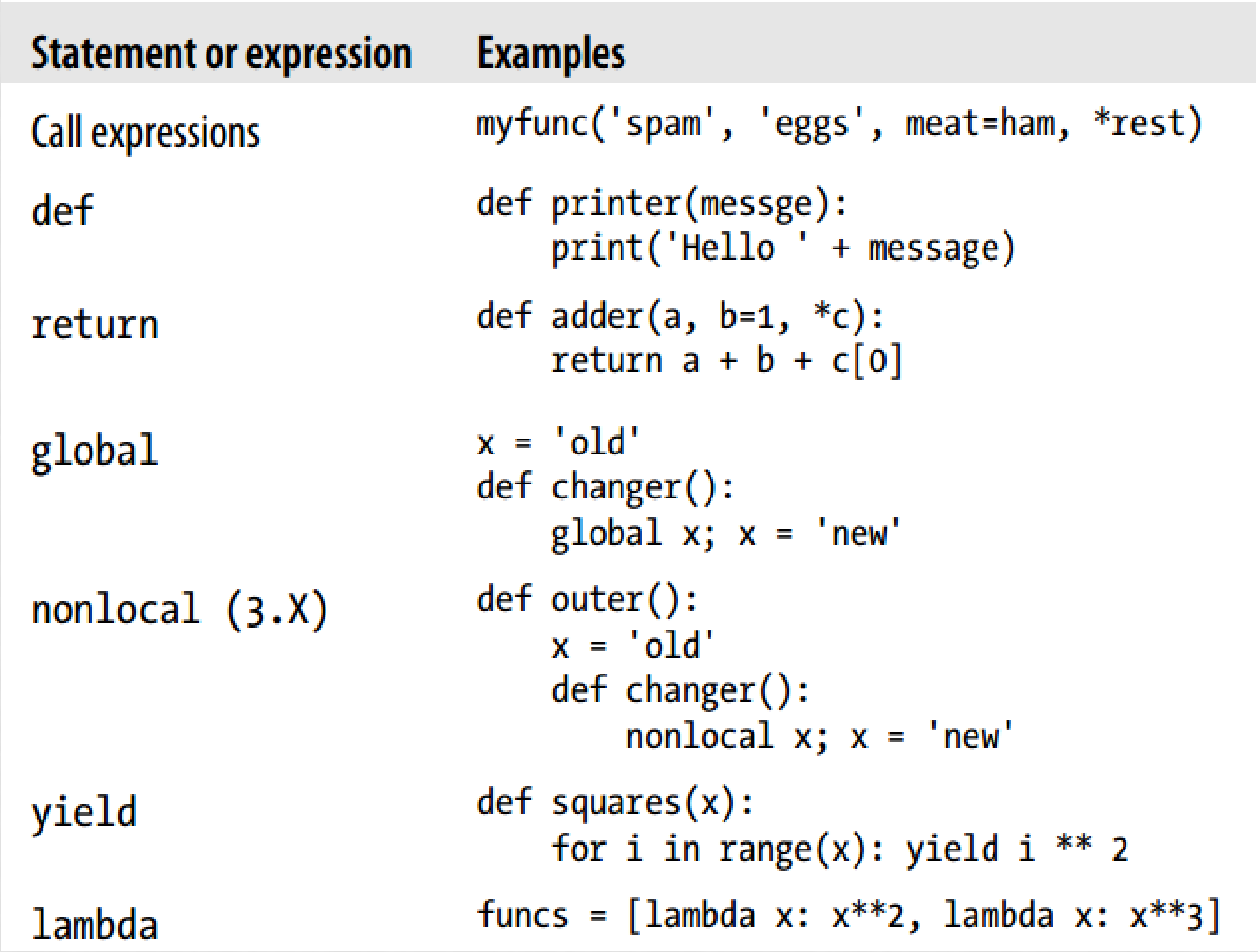
DEF

- this is sometimes called duck typing : "if it quacks like a duck it is a duck"
- checking is done runtime :



```
>>> def f(a, b):
        return(a * b)

>>> f('apple', 3)
'appleappleapple'
>>> f('apple', 'three')
Traceback (most recent call last):
  File "<pyshell#46>", line 1, in <module>
    f('apple', 'three')
  File "<pyshell#44>", line 2, in f
    return(a * b)
TypeError: can't multiply sequence by non-int of type 'str'
```

FUNCTION-RELATED EXPRESSIONS



Statement or expression	Examples
Call expressions	`myfunc('spam', 'eggs', meat=ham, *rest)`

`def`	`def printer(messge):`
	`    print('Hello ' + message)`

`return`	`def adder(a, b=1, *c):`
	`    return a + b + c[0]`

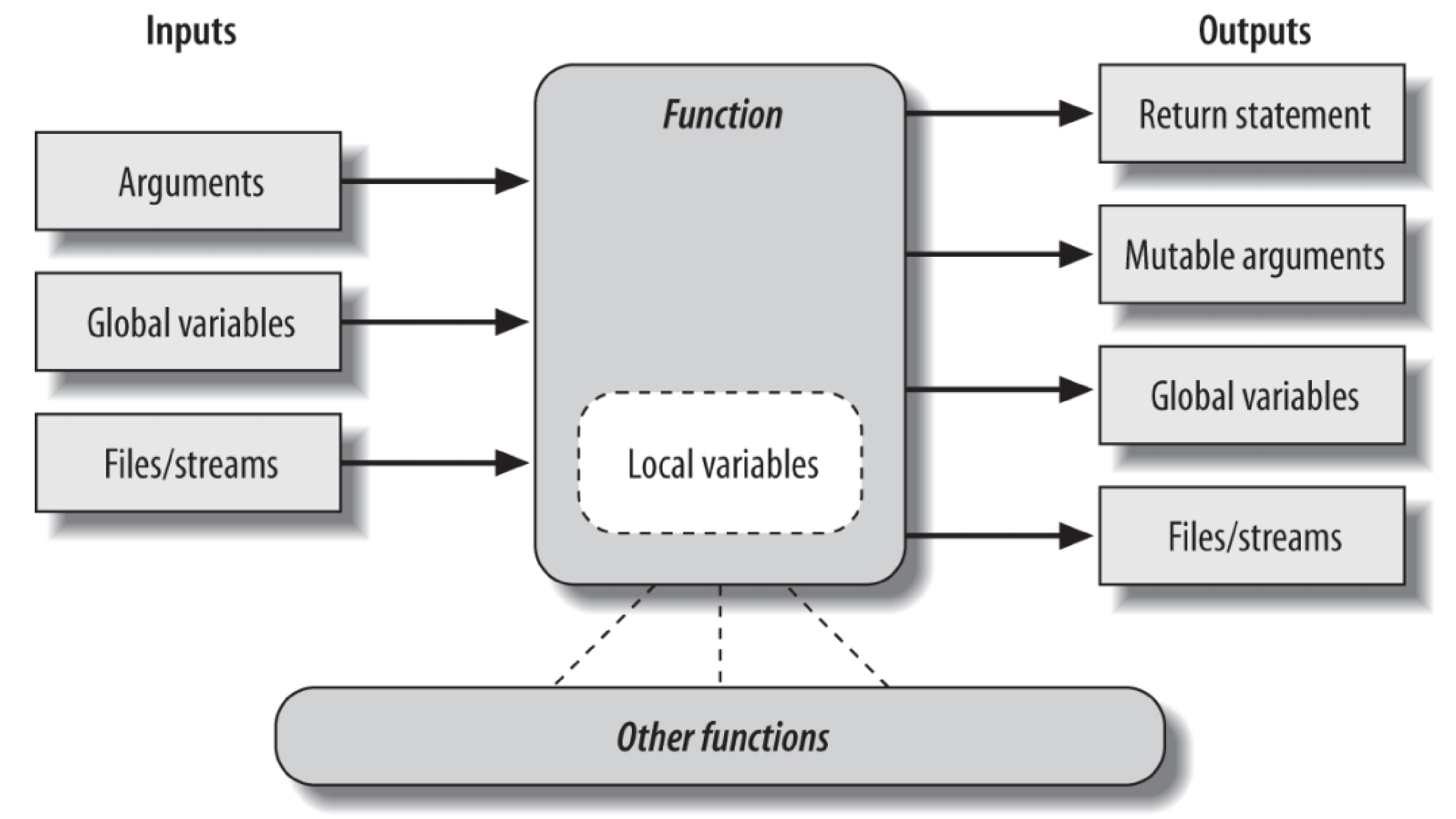
`global`	`x = 'old'`
	`def changer():`
	`    global x; x = 'new'`

`nonlocal (3.X)`	`def outer():`
	`    x = 'old'`
	`    def changer():`
	`        nonlocal x; x = 'new'`

`yield`	`def squares(x):`
	`    for i in range(x): yield i ** 2`

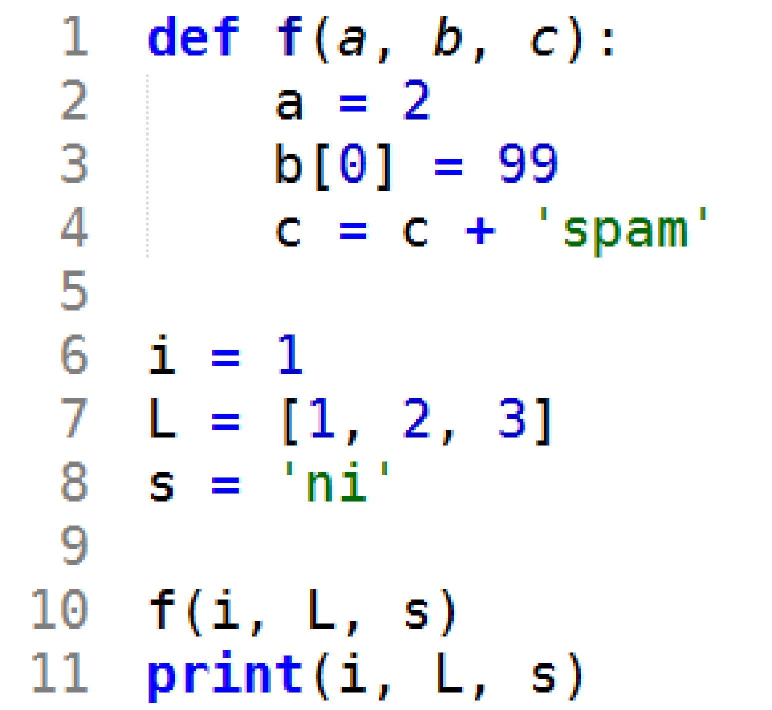
`lambda`	`funcs = [lambda x: x**2, lambda x: x**3]`

FUNCTION EXECUTION ENVIRONMENT



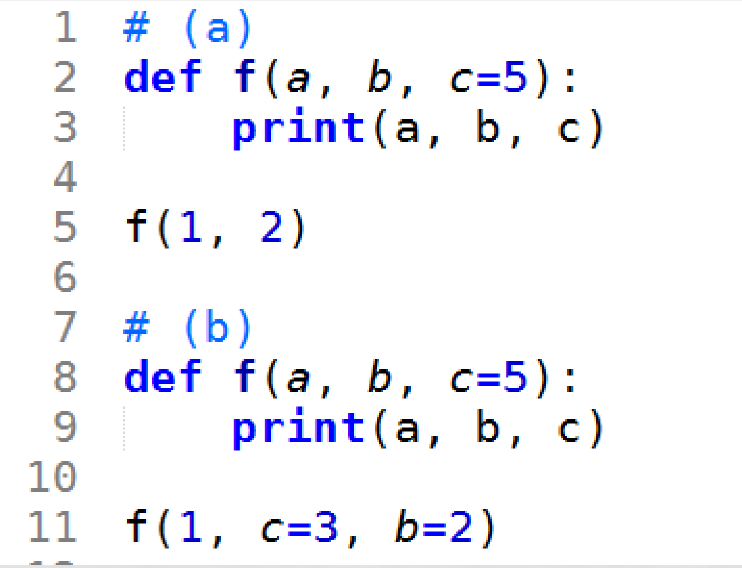
QUESTION

- all arguments are passed by reference
- only mutable objects can be changed in-place
- what will be printed?

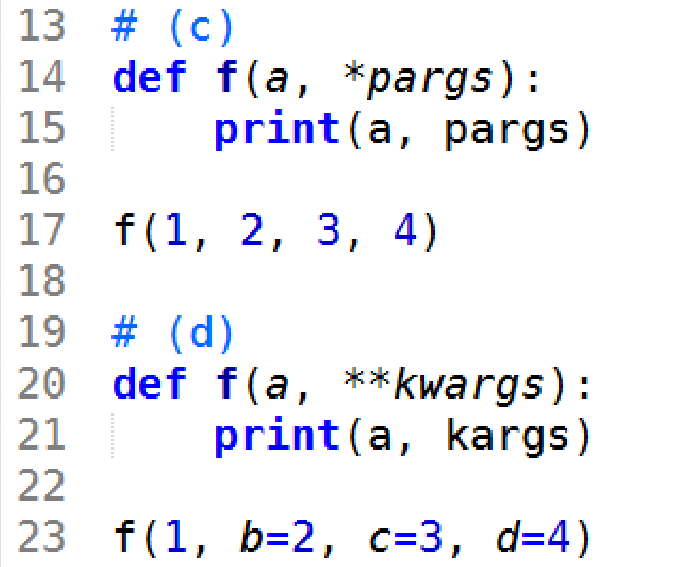


```python
def f(a, b, c):
    a = 2
    b[0] = 99
    c = c + 'spam'

i = 1
L = [1, 2, 3]
s = 'ni'

f(i, L, s)
print(i, L, s)
```

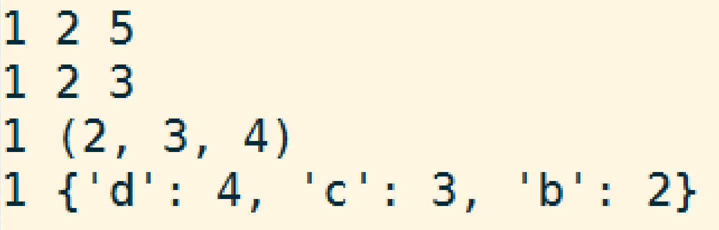
WHAT WILL BE PRINTED?



```python
# (a)
def f(a, b, c=5):
    print(a, b, c)

f(1, 2)

# (b)
def f(a, b, c=5):
    print(a, b, c)

f(1, c=3, b=2)
```



```python
# (c)
def f(a, *pargs):
    print(a, pargs)

f(1, 2, 3, 4)

# (d)
def f(a, **kwargs):
    print(a, kargs)

f(1, b=2, c=3, d=4)
```



```
1 2 5
1 2 3
1 (2, 3, 4)
1 {'d': 4, 'c': 3, 'b': 2}
```

ARGUMENT MATCHING

- by default, arguments are passed by position
- other options : match by name, provide default values, use collectors for extra arguments

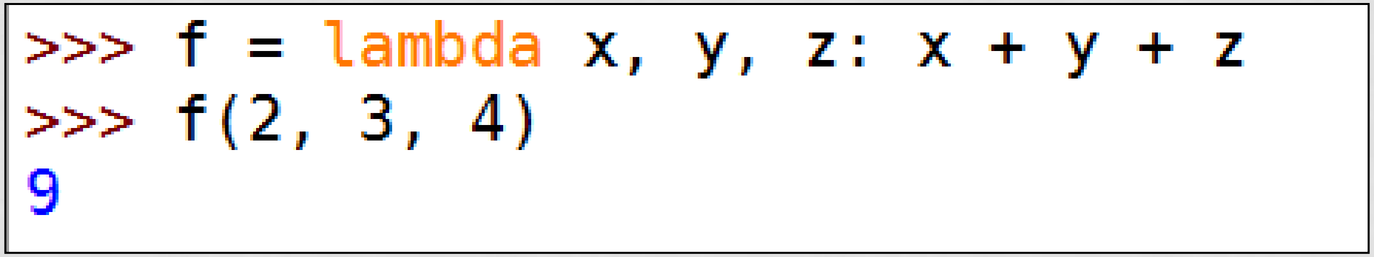
- **argument packing** when defining a function
  - `def func(*pargs) : collect positional arguments in a tuple`
  - `def func(**kwargs) : collect key-worded arguments in a dictionary`

- **argument un-packing** when calling a function
  - `func(*parg) : distribute arguments from a tuple`
  - `func(**kwarg) : distribute arguments from a dictionary`

LAMBDA : ANONYMOUS FUNCTION

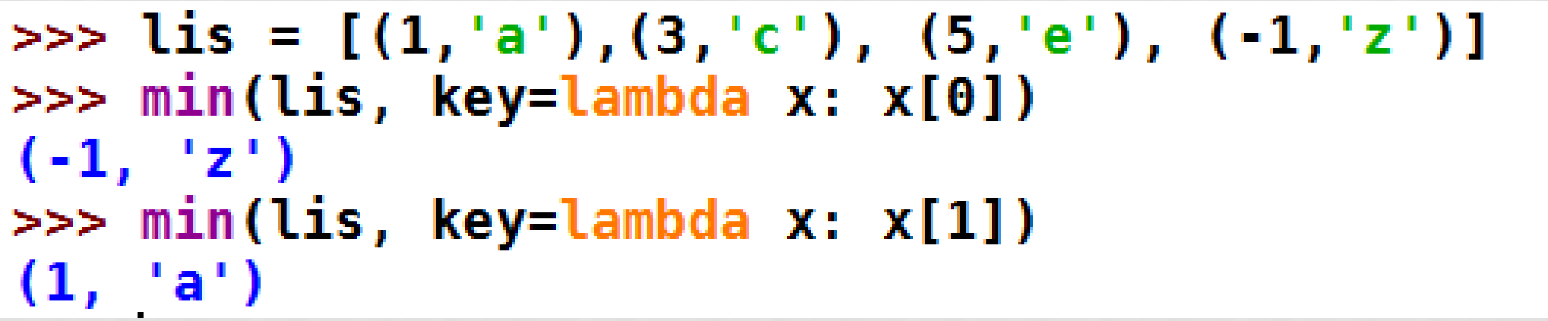
- lambda creates a function object (just like def) but doesn't assign a name to it
- lambda is '*syntactic suger*'
- lambda is an expression, not a statement
  - an expression returns a value
  - there is no return statement
- syntax :

```
lambda arg1, arg2: expression using arg1 & arg2
```

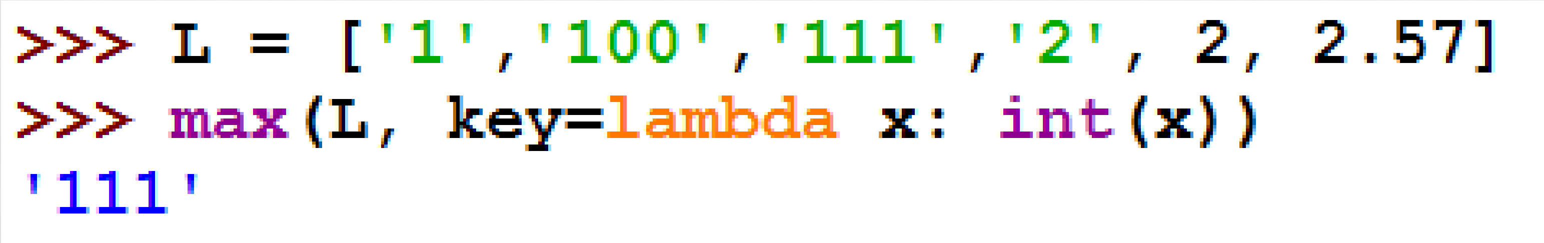


```
>>> f = lambda x, y, z: x + y + z
>>> f(2, 3, 4)
9
```

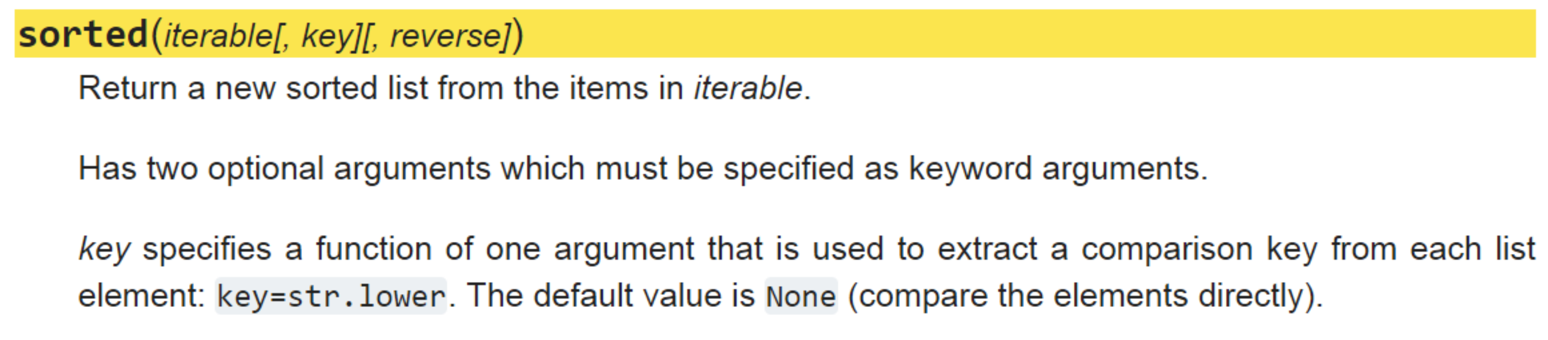
LAMBDA : ANONYMOUS FUNCTION



```
>>> lis = [(1,'a'),(3,'c'), (5,'e'), (-1,'z')]
>>> min(lis, key=lambda x: x[0])
(-1, 'z')
>>> min(lis, key=lambda x: x[1])
(1, 'a')
```



```
>>> L = ['1','100','111','2', 2, 2.57]
>>> max(L, key=lambda x: int(x))
'111'
```

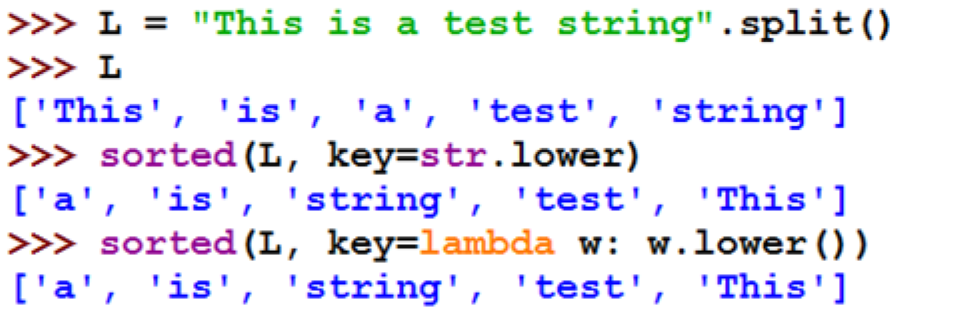


sorted(*iterable[, key][, reverse]*)

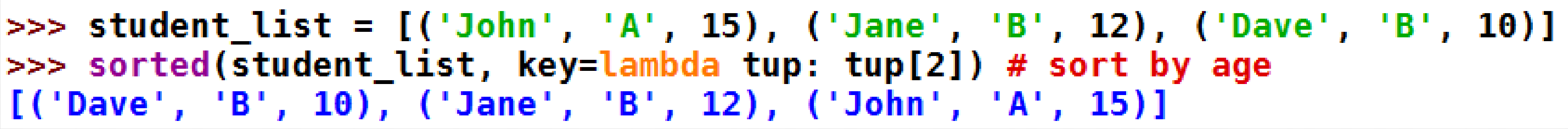
Return a new sorted list from the items in *iterable*.

Has two optional arguments which must be specified as keyword arguments.

key specifies a function of one argument that is used to extract a comparison key from each list element: `key=str.lower`. The default value is `None` (compare the elements directly).



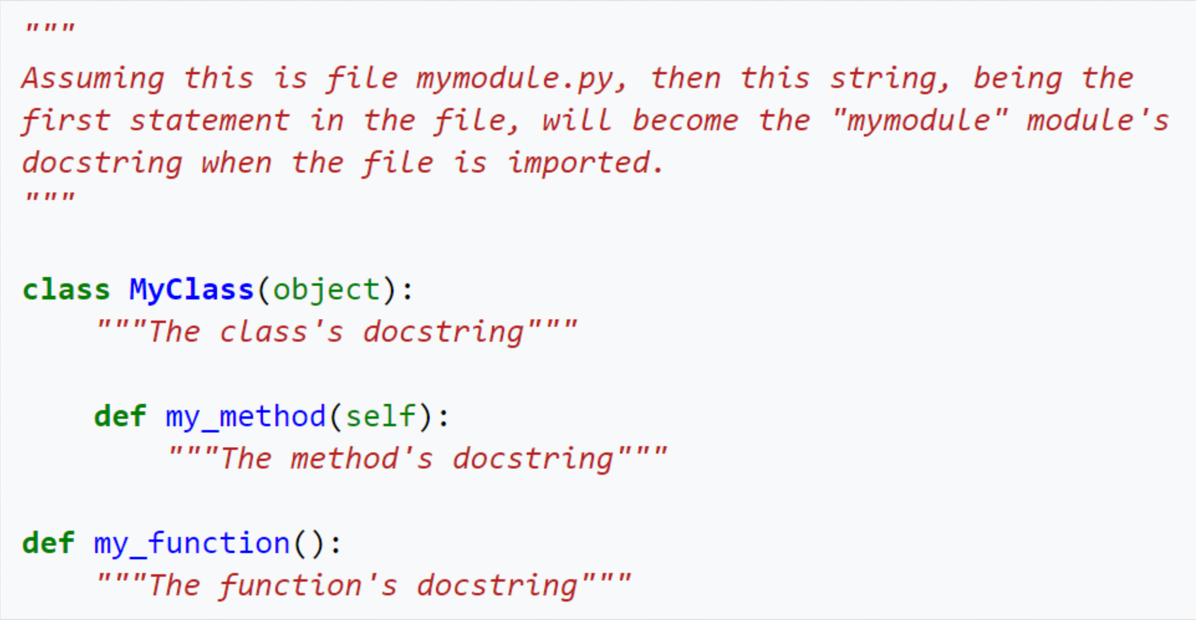
```
>>> L = "This is a test string".split()
>>> L
['This', 'is', 'a', 'test', 'string']
>>> sorted(L, key=str.lower)
['a', 'is', 'string', 'test', 'This']
>>> sorted(L, key=lambda w: w.lower())
['a', 'is', 'string', 'test', 'This']
```



```
>>> student_list = [('John', 'A', 15), ('Jane', 'B', 12), ('Dave', 'B', 10)]
>>> sorted(student_list, key=lambda tup: tup[2]) # sort by age
[('Dave', 'B', 10), ('Jane', 'B', 12), ('John', 'A', 15)]
```

DOCSTRING

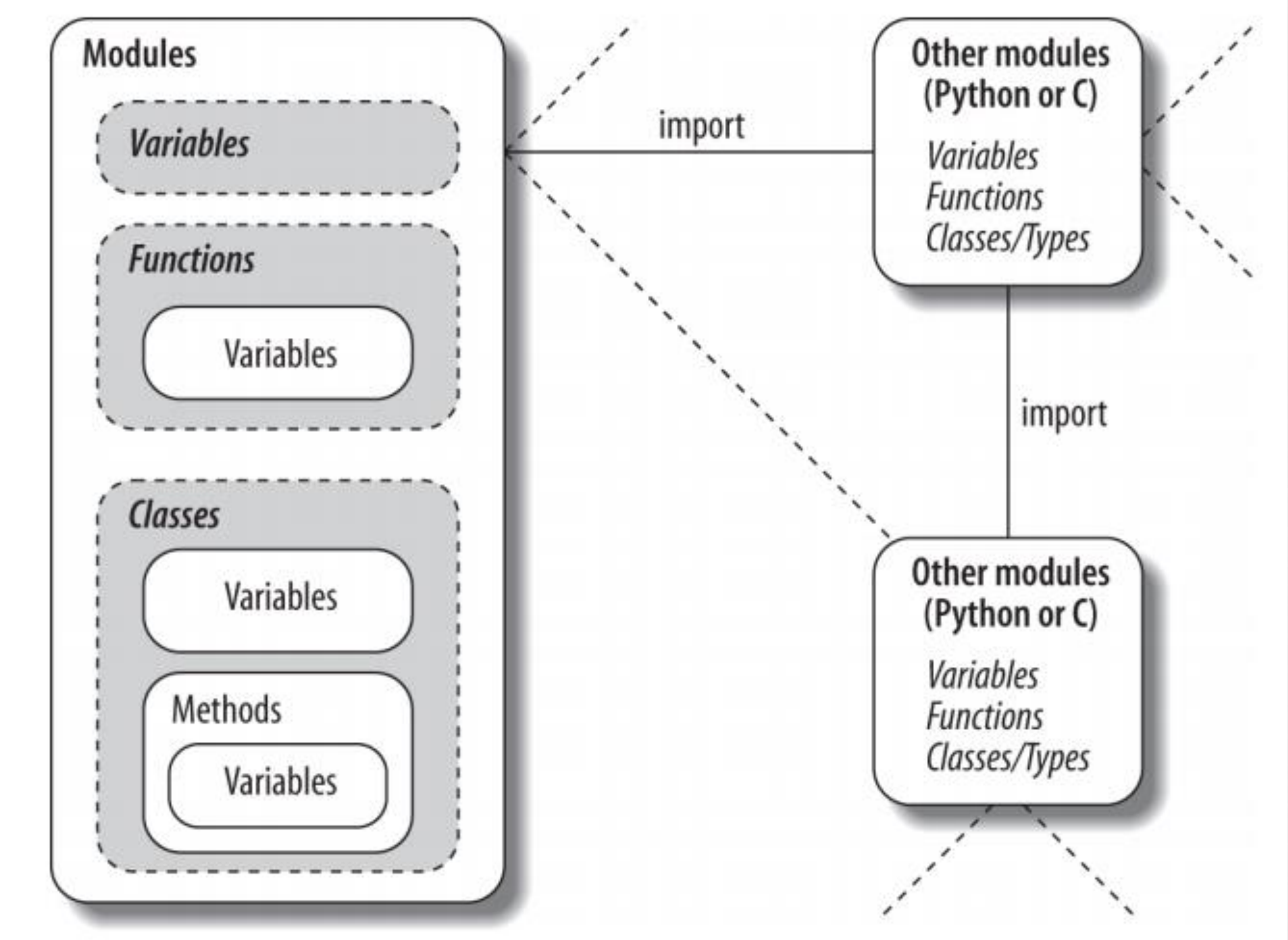
- # this is a single-line comment
- ''' this is a mulit-line comment or docstring '''
  - comments at the top of module files
  - or just below a def or class statement



```python
"""
Assuming this is file mymodule.py, then this string, being the
first statement in the file, will become the "mymodule" module's
docstring when the file is imported.
"""

class MyClass(object):
    """The class's docstring"""

    def my_method(self):
        """The method's docstring"""

def my_function():
    """The function's docstring"""
```

AGENDA

- tuples
- dictionaries
- sets
- functions
- **scope/visibility**
- comprehension
- modules and packages


Modules
Variables
Functions
Variables
Classes
Variables
Methods
Variables
import
Other modules (Python or C)
Variables
Functions
Classes/Types
import
Other modules (Python or C)
Variables
Functions
Classes/Types

SCOPE OR NAMESPACE

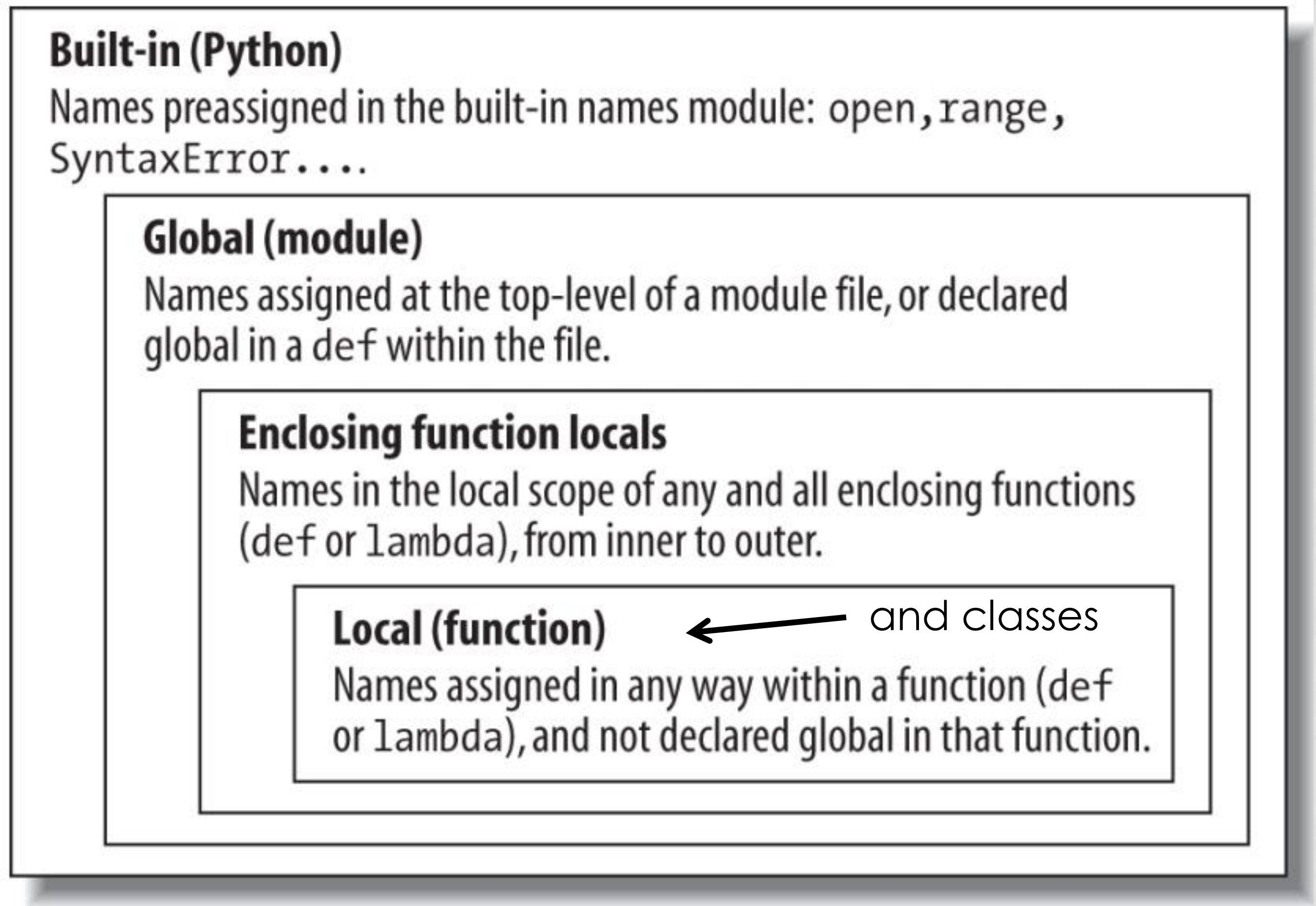
- the place where you assign a name in your source code determines the scope of a name
- a variable assigned (created)
  - outside a def or class has module scope and is global to the entire file
  - inside a def is has function scope and is local to that function
  - inside a class has class scope and is local to that class
- note : (unlike Java or C) there is no block-scope in Python, meaning control blocks like if, for and while don't count !

REFERRING TO VARIABLE X

- will look up x from inner scope to outer scope: function, enclosing function, global/module

- global : to refer to names that live in global scope
- nonlocal : to refer to names that live in an enclosing function scope (only relevant in case of nested functions)

THE L(E)GB SCOPE RULE



Built-in (Python)

Names preassigned in the built-in names module: open, range, SyntaxError....

Global (module)

Names assigned at the top-level of a module file, or declared global in a def within the file.

Enclosing function locals

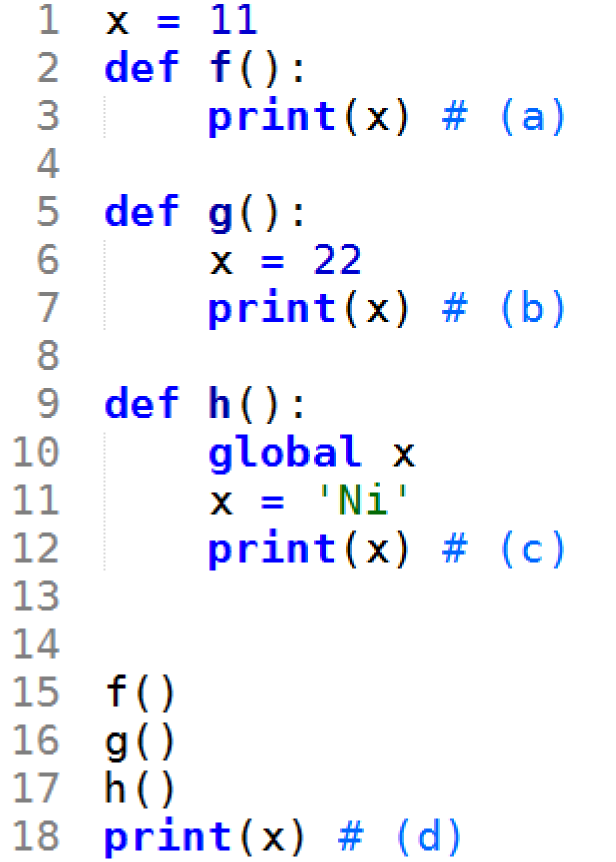
Names in the local scope of any and all enclosing functions (def or lambda), from inner to outer.

Local (function)

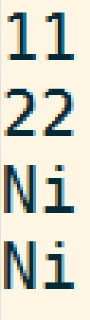
and classes

Names assigned in any way within a function (def or lambda), and not declared global in that function.

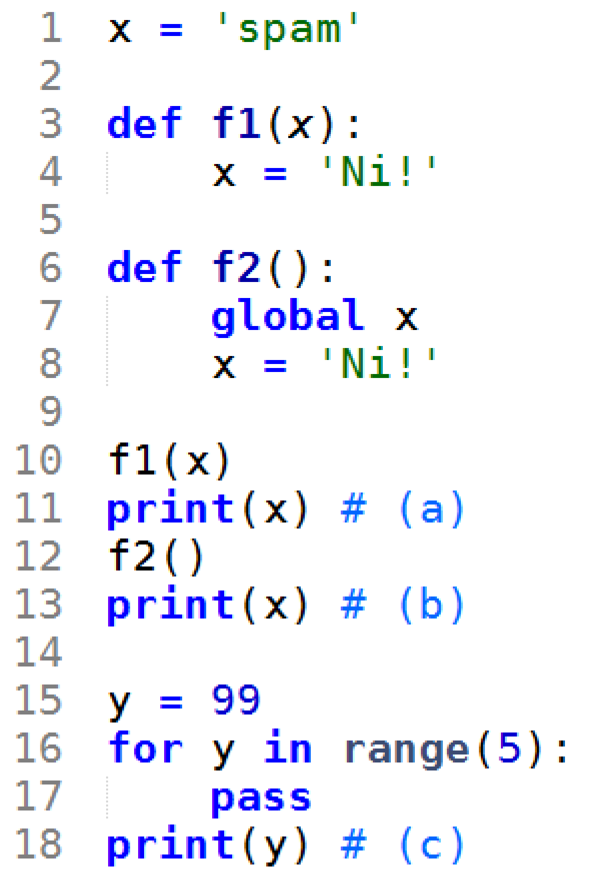
WHAT WILL BE PRINTED?



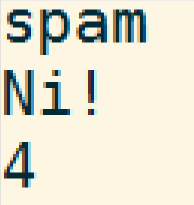
```python
x = 11
def f():
    print(x) # (a)

def g():
    x = 22
    print(x) # (b)

def h():
    global x
    x = 'Ni'
    print(x) # (c)


f()
g()
h()
print(x) # (d)
```



```
11
22
Ni
Ni
```



```python
x = 'spam'

def f1(x):
    x = 'Ni!'

def f2():
    global x
    x = 'Ni!'

f1(x)
print(x) # (a)
f2()
print(x) # (b)

y = 99
for y in range(5):
    pass
print(y) # (c)
```



```
spam
Ni!
4
```

AGENDA

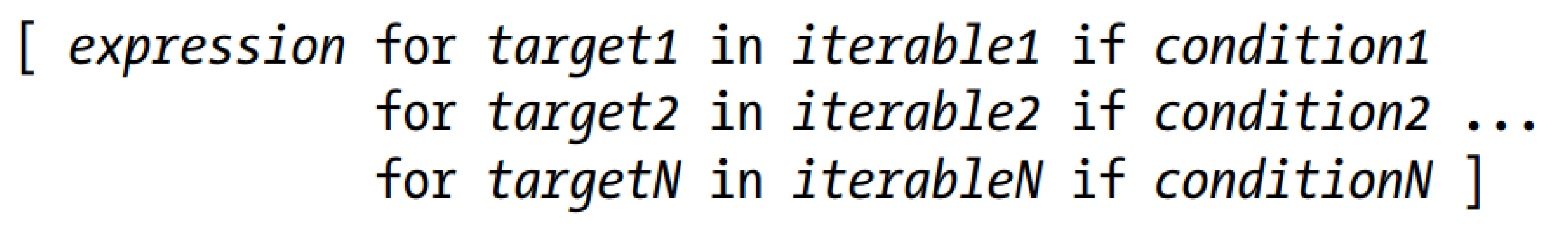
- tuples
- dictionaries
- sets
- functions
- scope/visibility
- **comprehension**
- modules and packages

COMPREHENSION

- inspired by functional programming language Haskell
- comprehensions applies an expressions to items in a list [] or dictionary {}
- in math :

$$S = \{ \underbrace{2 \cdot x}_{\text{output expression}} \mid \underbrace{x}_{\text{variable}} \in \underbrace{\mathbb{N}}_{\text{input set}} , \underbrace{x^2 > 3}_{\text{predicate}} \}$$

- in Python



```
[ expression for target1 in iterable1 if condition1
             for target2 in iterable2 if condition2 ...
             for targetN in iterableN if conditionN ]
```

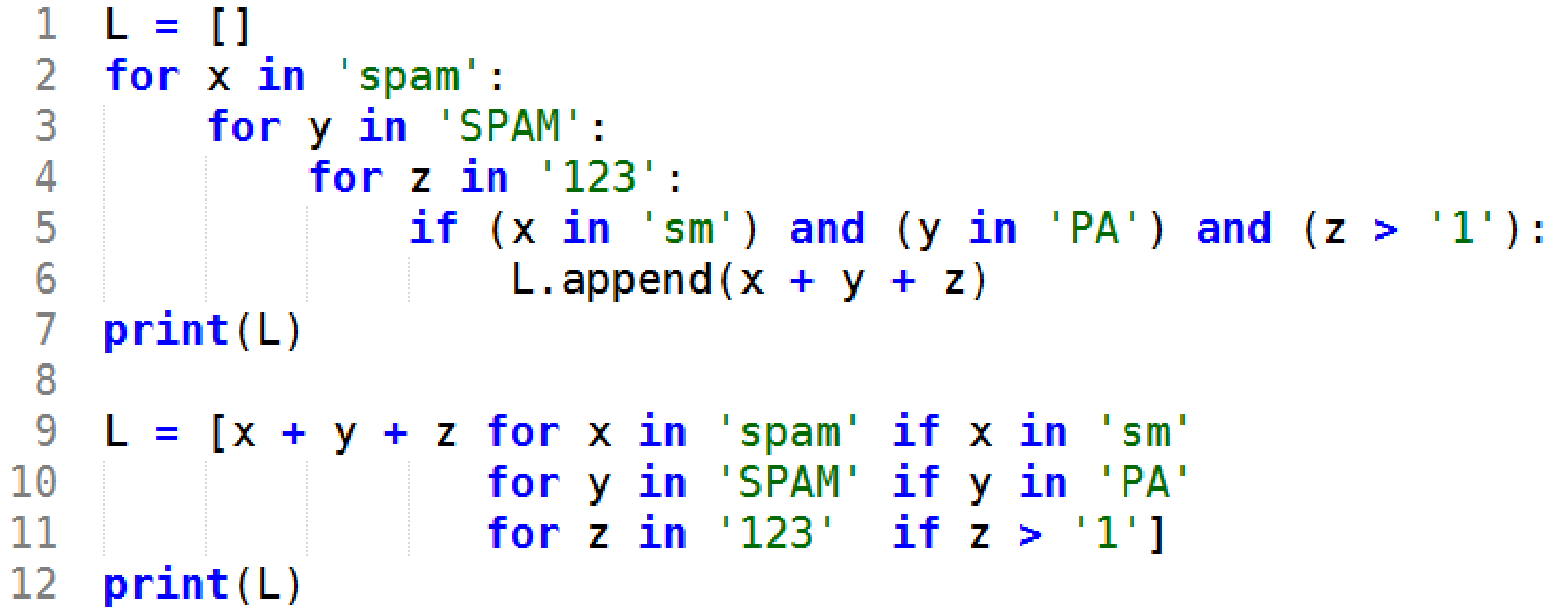
WHAT WILL BE PRINTED?

```
a) [0.5*x for x in range(5) if x < 1.5]

b) [int(x) for x in '12345']

c) s = 'abcd'
   [s[:i]+s[i:] for i in range(len(s))]

d) max(-x*x for x in range (-4,4))

e) {n: n**2 for n in range(4)}
```

ANSWERS

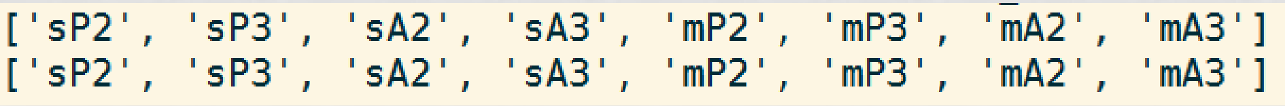
```
a) [0.0, 0.5]

b) [1, 2, 3, 4, 5]

c) ['abcd', 'abcd', 'abcd', 'abcd']

d) 0

e) {0: 0, 1: 1, 2: 4, 3: 9}
```

WHAT WILL BE PRINTED?



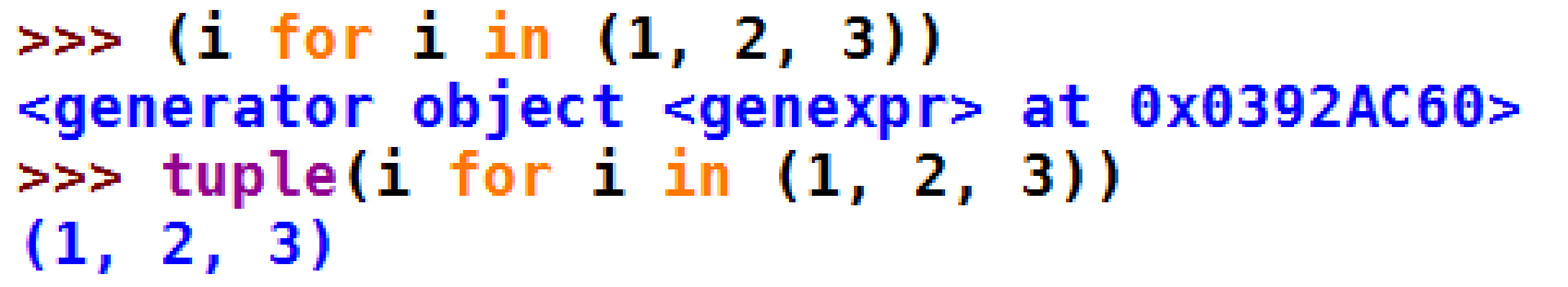
```python
L = []
for x in 'spam':
    for y in 'SPAM':
        for z in '123':
            if (x in 'sm') and (y in 'PA') and (z > '1'):
                L.append(x + y + z)
print(L)

L = [x + y + z for x in 'spam' if x in 'sm'
               for y in 'SPAM' if y in 'PA'
               for z in '123'  if z > '1']
print(L)
```

- results in 2*2*2 list elements (strings)



```
['sP2', 'sP3', 'sA2', 'sA3', 'mP2', 'mP3', 'mA2', 'mA3']
['sP2', 'sP3', 'sA2', 'sA3', 'mP2', 'mP3', 'mA2', 'mA3']
```

NO TUPLE COMPREHENSION



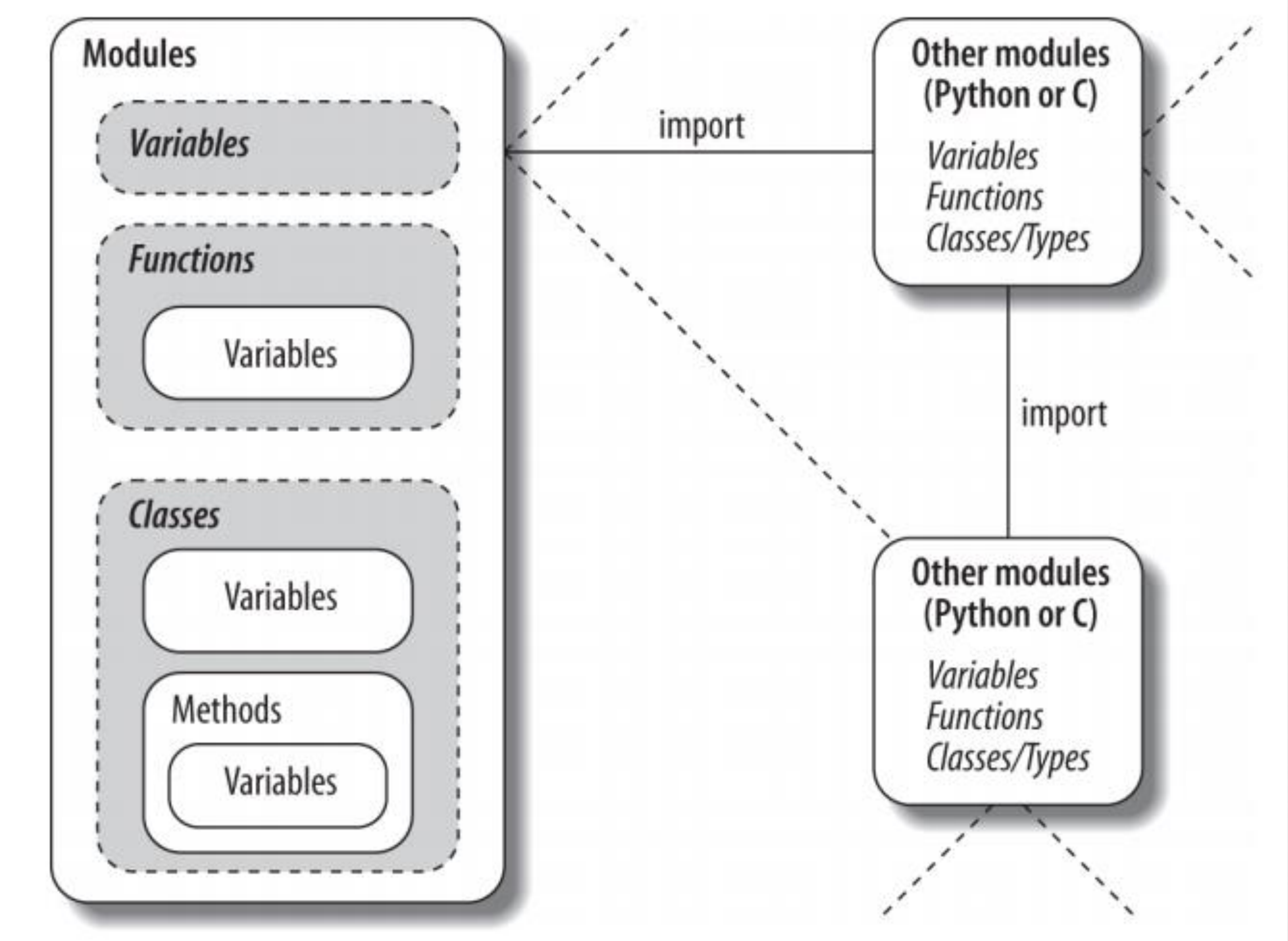
```
>>> (i for i in (1, 2, 3))
<generator object <genexpr> at 0x0392AC60>
>>> tuple(i for i in (1, 2, 3))
(1, 2, 3)
```

COMPREHENSION VS FOR LOOPS

- comprehension has often better performance
- comprehension is concise
- for loops make logic more explicit
- use comprehension only for simple iterations
  - avoid incomprehensible comprehensions ;-)

AGENDA

- tuples
- dictionaries
- sets
- functions
- scope/visibility
- comprehension
- **modules and packages**

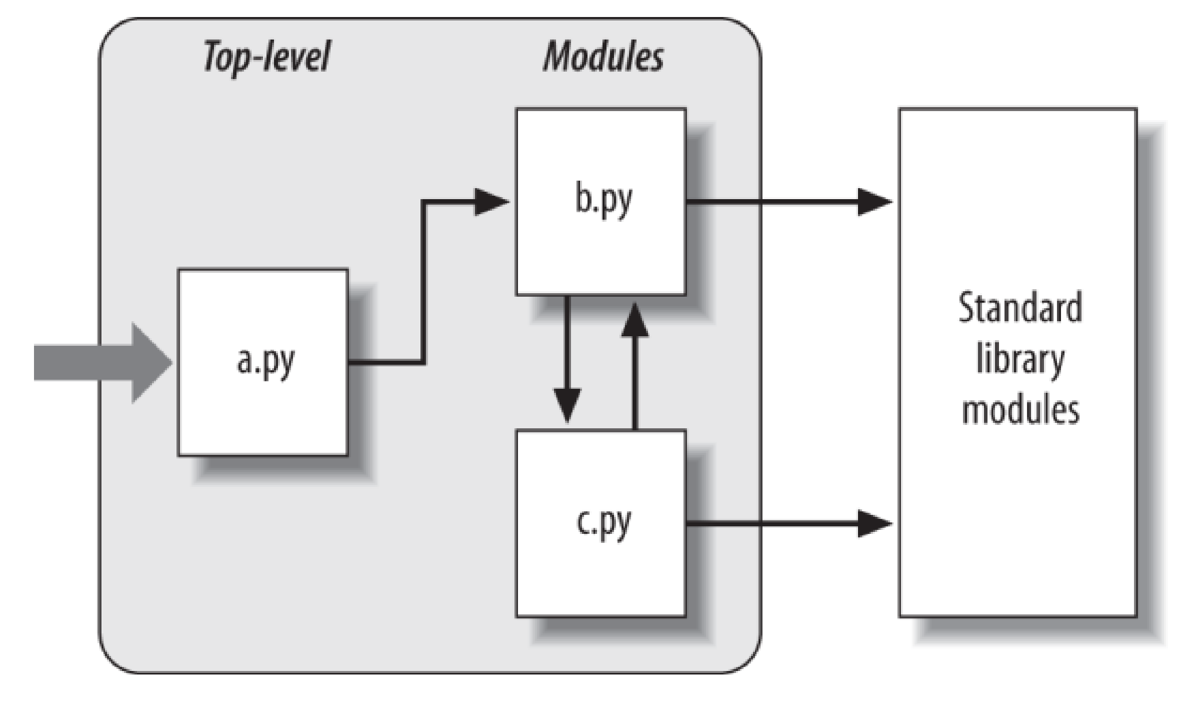

Modules
Variables
Functions
Variables
Classes
Variables
Methods
Variables
import
Other modules
(Python or C)
Variables
Functions
Classes/Types
import
Other modules
(Python or C)
Variables
Functions
Classes/Types

PYTHON PROGRAM

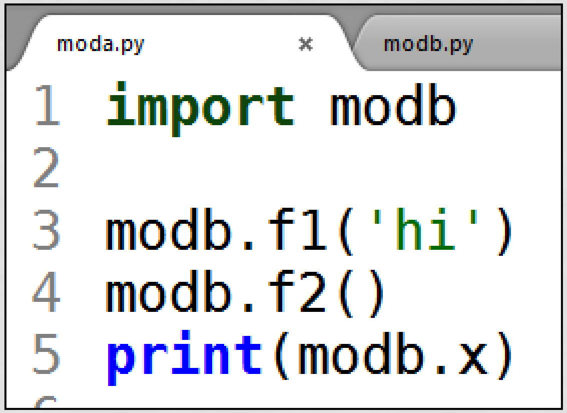
- Python program = a top-level file that you run + imported modules
  - you're always in a module, the file that you run gets the name __main__

3 WAYS TO IMPORT A MODULE

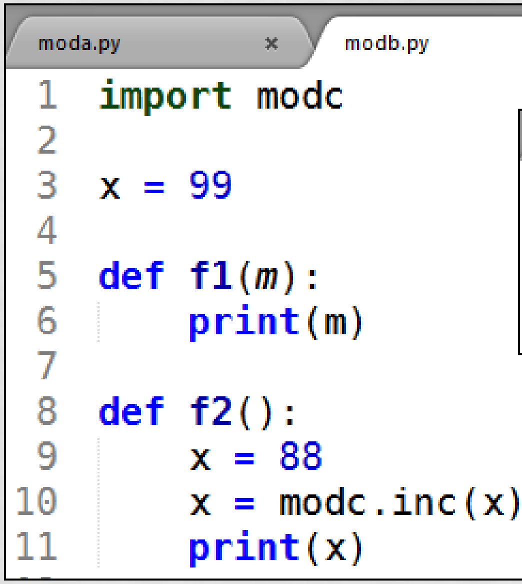
- **`import modb`**
  - gives access to *all* attributes of modb (variables, functions or classes)
  - you must refer to f2() as modb.f2()
- **`from modb import f1, f2, x`**
  - this will copy only b.f1, b.f2 and b.x
  - you can refer directly to f2() (instead of modb.f2())
- **`from modb import *`**
  - this will copy all attributes of modb
  - you can refer directly to f2() (instead of modb.f2())



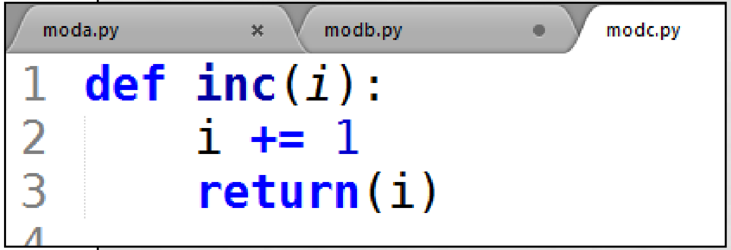
```
hi
89
99
```



```python
import modb

modb.f1('hi')
modb.f2()
print(modb.x)
```



```python
import modc

x = 99

def f1(m):
    print(m)

def f2():
    x = 88
    x = modc.inc(x)
    print(x)
```



```python
def inc(i):
    i += 1
    return(i)
```

IMPORT A MODULE

- **`import modb`** will compile it to byte code
  - byte code is stored byte code file (.pyc) in subdirectory _ _pycache_ _
- run de modb code to build the objects it defines
  - a top-level print() statement will actually show results
- every name assigned at top-level will become an attribute of modb, e.g. modb.x, modb.f1, modb,f2

IF __NAME__ == '__MAIN__'

- moda.py can be run as a standalone program, or can be imported as a module
- attribute _ _ name_ _ is set to _ _main_ _ if it runs as a top-level program file

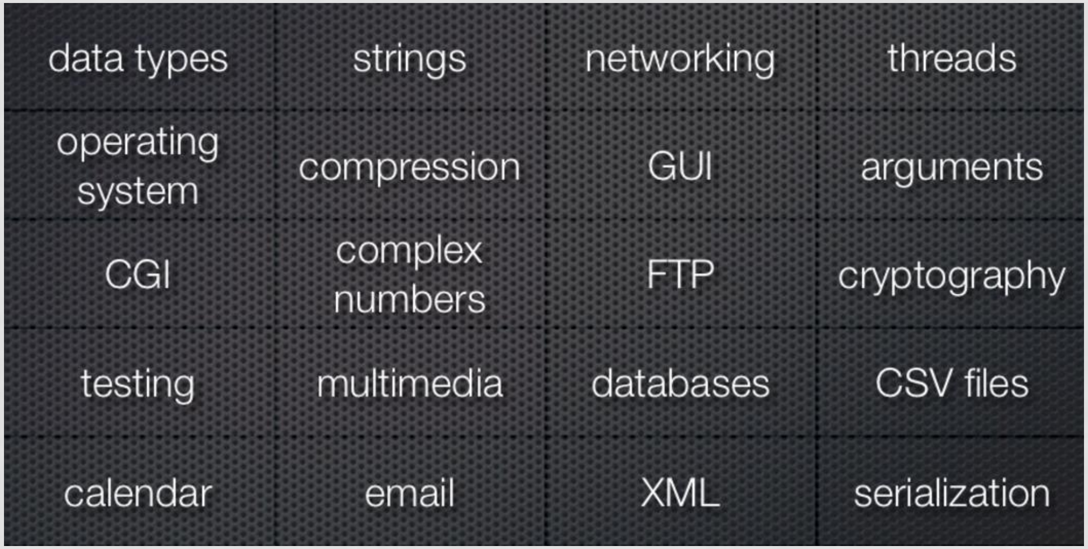
```
# at bottom of a file
if __name__ == '__main__':
    # do some testing and printing
```

NAMESPACES

- modules, functions and classes define namespaces
- modules correspond to files
- functions and classes live within a module
  - functions are created with ***def*** statements
  - classes are created with ***class*** statements

STANDARD LIBARY

- Python’s standard library is very extensive
  - access to system functionality such as file I/O
  - standardized solutions for many problems that occur in everyday programming
  - available on any standard installation



data types	strings	networking	threads
operating system	compression	GUI	arguments
CGI	complex numbers	FTP	cryptography
testing	multimedia	databases	CSV files
calendar	email	XML	serialization

INSTALLING PACKAGES

- Python distribution : an archive file that contains Python packages and modules (and possibly C extension packages)
- wheel : a built-package format for Python (ZIP-format archive with .whl extension)
- package manager : installation, upgrade and un-installation of Python packages
  - pip and conda are packet managers
- PyPI : the Python Package Index with >76000 packages

INSTALLING PACKAGES WITH PIP

- pip is part of Python 3
- Python packages only
- supports installing from PyPI
- $ pip install SomePackage # installs latest version
- will compile everything from source

PACKAGES

- package = a directory of Python code
  - similar to a Java package
- `import dir1.dir2.modb`
- `from dir1.dir2.modb import f1, f2, x`

- dir1 is in a rootdir, and rootdir must be in sys.path
- sys.path = the Python search path, try : import sys, print(sys.path)