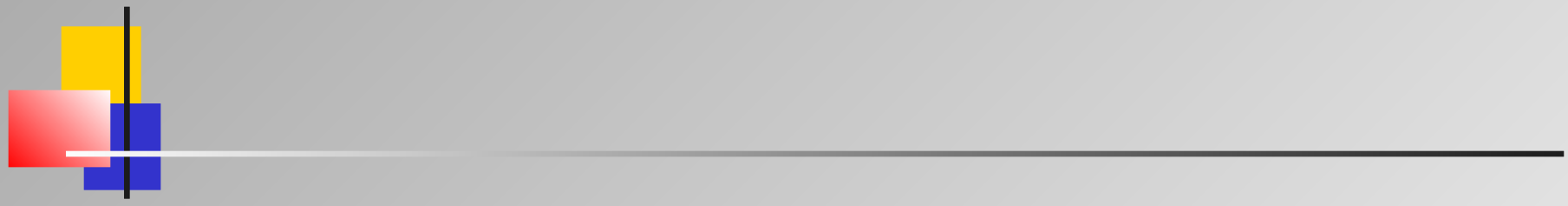
OOP and Scripting in Python

Python Advanced Features

Giuliano Armano - DMI Univ. di Cagliari

Python: Advanced Features

- Callables
- Iterators and Generators
- Functional programming



Callables

Python Advanced Features: Callables

Callables

- Types that support the function call operation are named “callable”
- List of “callable” types:
  - Functions YES
  - Methods YES
  - Types (e.g., tuples, lists, dictionaries) YES
  - Class instances (supporting `__call__`) YES

Callables (e.g., list-to-dict)

```
>>> q = [('x',1),('y',2),('z',3)]
>>> q
[('x', 1), ('y', 2), ('z', 3)]
>>> dict(q)
{'y': 2, 'x': 1, 'z': 3}
>>>
```

Callables: Function Objects - I

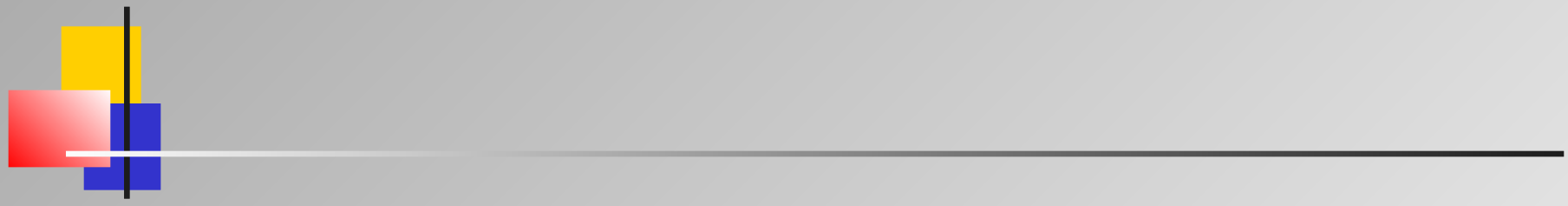
```
>>> class Callable(object):
...   def __init__(self,function):
...     self.function = function
...   def __call__(self,*args, **kwargs):
...     return self.function(*args, **kwargs)
...

>>> def inc(x=0, delta=1):
...   return x+delta
...

>>> INC = Callable(inc)
>>> INC(34)
35
```

Callables: Function Objects - II

```
>>> class Callable(object):
...   def __init__(self,function):
...     self.function = function
...     self.numCalls = 0
...   def __call__(self,*args, **kwargs):
...     self.numCalls += 1
...     return self.function(*args, **kwargs)
...

>>> def foo(x=0, y=0):
...   return (x+1,y+1)
...
```

Callables: Function Objects - II

```
>>> FOO = Callable(foo)
>>> z, w = FOO(3,4)
>>> FOO.numCalls
1
>>> FOO(33,44)
(34,45)
>>> FOO.numCalls
2
```



Iterators

Python Advanced Features: Iterators

Iterators

- Iterators are standard tools for iterating over a collection (string, tuple, list, dictionary)
- Iterators can be used also for iterating on instances
- In any case, when the iteration reached its end, a StopIteration exception is raised
- The module itertools contains useful iterators

Any “for” statement actually uses an iterator to perform iteration (and StopIteration forces a “break”)

Iterating over a Sequence (string)

```
>>> it = iter('abc')
>>> it.__next__()
a
>>> it.__next__()
b
>>> it.__next__()
c
>>> it.__next__()

Traceback (most recent call last):
File "<pyshell#493>", line 2, in -toplevel- print it.next()
StopIteration
>>>
```

Iterating over a Sequence (string)

How to avoid the StopIteration exception ...

```
>>> it = iter('abc')
>>> try:
...   while True:
...     print(next(it))
... except StopIteration:
...   print('End Iteration')
...

a
b
c
End Iteration
>>>
```

Iterating over a Sequence (list)

```
>>> it = iter([1,2,'a'])
>>> while True:
...     print(it.__next__())
...

1
2
a

Traceback (most recent call last):
File "<pyshell#493>", line 2, in -toplevel- print it.next()
StopIteration
>>>
```

Iterators (on objects) - 1a

Using delegation to perform iteration

```
>>> class Counter(object):
...   def __init__(self):
...     self.cnt = -1
...   def __call__(self):
...     self.cnt += 1
...     return self.cnt
...
```

Any iterator built upon an instance of Counter actually delegates __call__ to perform the actual computation

Iterators (on objects) - 1b

```
>>> c = Counter()
>>> it = iter(c,5)
>>> it.__next__()              # same as: c.__call__()
0
>>> it.__next__()              # same as: c.__call__()
1
>>> c.__call__()
2
```

Iterators (on objects) - 1c

Using delegation to perform iteration

```
>>> it = iter(Counter(),5)
>>> while True:
...     print(it.__next__())
...


0
1
2
3
4
Traceback (most recent call last):
File "<pyshell#476>", line 2, in -toplevel- print
   it.next()
StopIteration
```

when it.next() returns 5 a StopIteration is raised

Iterators (on objects) - 1d

Using delegation to perform iteration

```
>>> it = iter(Counter(),5)
>>> for x in it:
...     print(x)
...

0
1
2
3
4
>>>
```

Iterators (on objects) - 2a

Any object can be made "iterable"

```
>>> class Counter(object):
...     def __init__(self, maxvalue):
...         self.maxvalue = maxvalue
...     def __iter__(self):
...         self.cnt = -1
...         return iter(self,self.maxvalue)
...     def __call__(self):
...         self.cnt += 1
...         return self.cnt
...
```

On creation, the iterator delegates __iter__ to return a valid "iterable" sequence

Sentinel!!!

Iterators (on objects) - 2b

```
>>> for x in Counter(5):
...   print(x)
...

0
1
2
3
4
>>>
```

Iterators (itertools)

`from itertools import *`

- Some itertools:

```
chain (*iterables)
count(n=0)
cycle(iterable)
imap(function, *iterables)

... etc. ...
```

Itertools (chain)

from itertools import *

```
>>> for x in chain([1,2,3],['a','b','c']):
...     print(x)
...

1
2
3
a
b
c
>>>
```

Itertools (count)

from itertools import *

```
>>> for x in count():
...     print(x)
...

0
1
2
3
4
5
... etc. ...
```

Itertools (count)

... equivalent to itertools.count

```
>>> def count(n=0):
...   while True:
...     yield n; n += 1
...

>>> for x in count():
...   print(x)
...

0
1
... etc. ...
```

Actually, a generator ...

Itertools (cycle)

from itertools import *

```
>>> for x in cycle([1,2,3]):
...     print(x)
...

1
2
3
1
2
3
... etc. ...
```

Itertools (imap)

from itertools import *

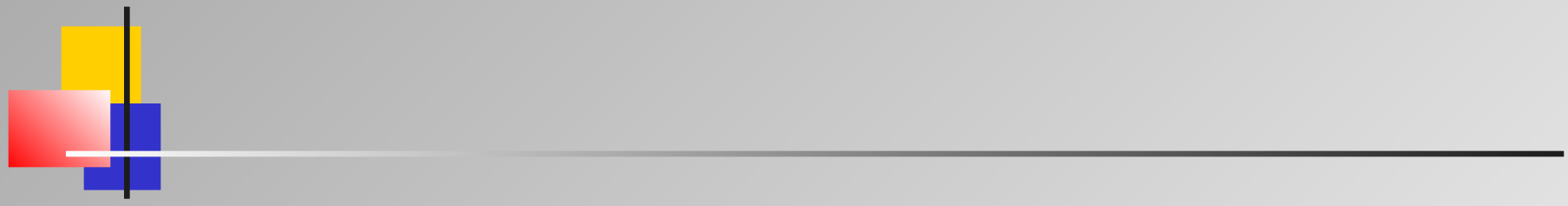
```
>>> it = imap(lambda x,y: x+y,[1,2,3],[4,5,6,7,8,9])
>>> it.next()
5
>>> it.next()
7
>>> it.next()
9
>>> it.next()

Traceback (most recent call last):
File "<pyshell#22>", line 1, in -toplevel- it.next()
StopIteration
```

Generators

```
>>> def foogen(sentinel):
      count = 0
      while True:
        yield count; count +=1
        if count == sentinel: break

>>> for x in foogen(4):
      print(x)

0
1
2
3
```

Generator Expressions

```
>>> qqq = (x for x in [10,20,30,40] if x < 25 )
>>> for a in qqq:
      print(a)

10
20

>>> [x for x in [10,20,30,40] if x < 25 ]
[10, 20]
>>>
```



Functional Programming

Python Advanced Features: Functional Programming

Functional Programming

- Lambda (anonymous) functions YES
- Call function by name YES
- Function composition YES
- Sequence processing (map, filter, reduce) YES

Lambda (Anonymous) Functions

```
>>> def incgen(y=1):
...     return lambda x: x+y
...

>>> inc1 = incgen()
>>> inc2 = incgen(2)
>>> inc1(10)
11
>>> inc2(10)
12
>>>
```

Call Function by Name

```
>>> def add(*numbers):
...     res = 0
...     for x in numbers:
...         res += x
...     return res
...

>>> add(1,2,3,4)
10
>>> apply(add,[1,2,3,4])       # deprecated!
10
>>>
```

Function Composition

```
>>> def compose(f1,f2):
...   return lambda x: f1(f2(x))
...

>>> lsqrt = compose(log,sqrt)
>>> lsqrt(10)
1.151292546497023
>>> log(sqrt(10))
1.151292546497023
>>>
```

Sequence Processing: map

```
>>> a = ['x','y','z']
>>> b = [1,2,3]
>>> w = map(lambda x,y: (x,y),a,b)
>>> w
[('x', 1), ('y', 2), ('z', 3)]
>>> dict(w)
{'y': 2, 'x': 1, 'z': 3}
>>> def myDict(keyws,values):
      return dict(map(lambda x,y: (x,y),keyws,values))

>>> aaa = myDict(['a', 'b'],[1,2])
>>> aaa
{'a': 1, 'b': 2}
>>>
```

Sequence Processing: map

```
>>> def add10(x):
...     return x+10
...

>>> map(add10,[10,20,30,40])
[20, 30, 40, 50]
>>>
```

Sequence Processing: filter

```
>>> filter(lambda x: x < 35,[10,20,30,40])
[10, 20, 30]

>>> [x for x in [10,20,30,40] if x < 35 ]
[10, 20, 30]
>>>
```

Sequence Processing: reduce

```
>>> reduce(lambda x,y: x+y,[1,2,3,4])
10
>>> ((1+2)+3)+4
10
>>> def logsin(x,y):
...    return log(abs(x)) * sin(y)
...

>>> reduce(logsin,[10,20,30])
-0.7340611369093767
>>> logsin(logsin(10,20),30)
-0.7340611369093767
```