301AA - Advanced Programming

Lecturer: Andrea Corradini
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

Course pages:
http://pages.di.unipi.it/corradini/Didattica/AP-18/

AP-2017-28: Even More on Python
We have seen:

- Basic and Sequence Datatypes
- Dictionaries
- Control Structures
- List Comprehension
- Function definition
- Positional and keyword arguments of functions

- Namespaces and Scopes
- Object Oriented programming in Python
- Inheritance
- Iterators and generators
- Functions as objects
- Higher-order functions
- Importing modules
Next topics

• More on higher-order functions
• Decorators
• Garbage collection and GIL
• Other criticisms to Python
• Exceptions in Python... in 2 slides!
Higher-order functions

• Functions can be passed as argument and returned as result
• Main combinators (map, filter) predefined: allow standard functional programmin style in Python
• Heavy use of iterators, which support laziness
• Lambdas supported for use with combinators

  `lambda arguments: expression`  
  – The body can only be a single expression
Map

```python
>>> print(map.__doc__)  # documentation
map(func, *iterables) --> map object
Make an iterator that computes the function using arguments from each of the iterables. Stops when the shortest iterable is exhausted.

>>> map(lambda x:x+1, range(4))  # lazyness: returns <map object at 0x10195b278>  # an iterator
>>> list(map(lambda x:x+1, range(4)))
[1, 2, 3, 4]
>>> list(map(lambda x, y : x+y, range(4), range(10)))
[0, 2, 4, 6]  # map of a binary function
>>> z = 5  # variable capture
>>> list(map(lambda x : x+z, range(4)))
[5, 6, 7, 8]
```
Map and List Comprehension

- List comprehension can replace uses of `map`

```python
>>> list(map(lambda x:x+1, range(4)))
[1, 2, 3, 4]
>>> [x+1 for x in range(4)]
[1, 2, 3, 4]
>>> list(map(lambda x, y : x+y, range(4), range(10)))
[0, 2, 4, 6]  % map of a binary function
>>> [x+y for x in range(4) for y in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5,...  % NO!
>>> [x+y for (x,y) in zip(range(4),range(10))]  % OK
[0, 2, 4, 6]
>>> print(zip.__doc__)
zip(iter1 [,iter2 [...]]) --> zip object
Return a zip object whose .__next__() method returns a tuple where the i-th element comes from the i-th iterable argument. The .__next__() method continues until the shortest iterable in the argument sequence is exhausted and then it raises StopIteration.
```
Filter (and list comprehension)

```python
>>> print(filter.__doc__)  # documentation
filter(function or None, iterable) --> filter object
Return an iterator yielding those items of iterable for which function(item) is true. If function is None, return the items that are true.

>>> filter(lambda x: x % 2 == 0, [1, 2, 3, 4, 5, 6])
<filter object at 0x102288a58>  # lazyness
>>> list(_)
[2, 4, 6]

>>> [x for x in [1, 2, 3, 4, 5, 6] if x % 2 == 0]
[2, 4, 6]  # same using list comprehension

% How to say "false" in Python

>>> list(filter(None,
               [1, 0, -1, '', "Hello", None, [], [1], (), True, False]))
[1, -1, 'Hello', [1], True]
```
More modules for functional programming in Python

- **functools**: Higher-order functions and operations on callable objects, including:
  - `reduce(function, iterable[, initializer])`

- **itertools**: Functions creating iterators for efficient looping. Inspired by constructs from APL, Haskell, and SML.
  - `count(10) --> 10 11 12 13 14 ...`
  - `cycle('ABCD') --> A B C D A B C D ...`
  - `repeat(10, 3) --> 10 10 10`
  - `takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4`
  - `accumulate([1,2,3,4,5]) --> 1 3 6 10 15`
Decorators

• A **decorator** is any callable Python object that is used to modify a **function**, method or class definition.

• A decorator is passed the original object being defined and returns a modified object, which is then bound to the name in the definition.

• (Function) Decorators exploit Python **higher-order features**:
  – Passing functions as argument
  – Nested definition of functions
  – Returning function

• Widely used in Python (system) programming

• Support several features of meta-programming
Basic idea: wrapping a function

```python
def my_decorator(func):    # function as argument
    def wrapper():    # defines an inner function
        print("Something happens before the function.")
        func()    # that calls the parameter
        print("Something happens after the function.")
    return wrapper    # returns the inner function

def say_hello():    # a sample function
    print("Hello!"
# 'say_hello' is bound to the result of my_decorator
say_hello = my_decorator(say_hello)    # function as arg
>>> say_hello()    # the wrapper is called
Something happens before the function.
Hello!
Something happens after the function.
```
Syntactic sugar: the "pie" syntax

```python
def my_decorator(func):    # function as argument
def wrapper():            # defines an inner function
    ... # as before
return wrapper # returns the inner function

def say_hello():          # HEAVY! 'say_hello' typed 3x
    print("Hello!")
say_hello = my_decorator(say_hello)
```

- Alternative, equivalent syntax

```python
@my_decorator
def say_hello():
    print("Hello!")
```
def do_twice(func):
    def wrapper_do_twice():
        func()  # the wrapper calls the argument twice
        func()
    return wrapper_do_twice

@do_twice
def say_hello():  # a sample function
    print("Hello!")

>>> say_hello()  # the wrapper is called
Hello!
Hello!

@do_twice  # does not work with parameters!!
def echo(str):  # a function with one parameter
    print(str)

>>> echo("Hi...")  # the wrapper is called
TypErr: wrapper_do_twice() takes 0 pos args but 1 was given
>>> echo()
TypErr: echo() missing 1 required positional argument: 'str'
do_twice for functions with parameters

• Decorators for functions with parameters can be defined exploiting *args and **kwargs

```python
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        func(*args, **kwargs)
        return wrapper_do_twice

@do_twice
def say_hello():
    print("Hello!")

>>> say_hello()
Hello!
Hello!
```

```python
@do_twice
def echo(str):
    print(str)

>>> echo("Hi... ")
Hi...
Hi...
```
General structure of a decorator

• Besides passing arguments, the wrapper also forwards the result of the decorated function
• Supports introspection redefining **name** and **doc**

```python
import functools
def decorator(func):
    @functools.wraps(func)  # supports introspection
    def wrapper_decorator(*args, **kwargs):
        # Do something before
        value = func(*args, **kwargs)
        # Do something after
        return value
    return wrapper_decorator
```
import functools
import time

def timer(func):
    """Print the runtime of the decorated function""
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()
        value = func(*args, **kwargs)
        end_time = time.perf_counter()
        run_time = end_time - start_time
        print(f"Finished {func.__name__!r} in {run_time:.4f} secs")
        return value

return wrapper_timer

@timer
def waste_some_time(num_times):
    for _ in range(num_times):
        sum([i**2 for i in range(10000)])
Other uses of decorators

- **Debugging**: prints argument list and result of calls to decorated function
- **Registering plugins**: adds a reference to the decorated function, without changing it
- In a web application, can wrap some code to check that the user is logged in
- **@staticmethod and @classmethod** make a function invocable on the class name or on an object of the class
- More: decorators can be nested, can have arguments, can be defined as classes...
import functools
from decorators import count_calls

def cache(func):
    """Keep a cache of previous function calls""
    @functools.wraps(func)
    def wrapper_cache(*args, **kwargs):
        cache_key = args + tuple(kwargs.items())
        if cache_key not in wrapper_cache.cache:
            wrapper_cache.cache[cache_key] = func(*args, **kwargs)
        return wrapper_cache.cache[cache_key]
    wrapper_cache.cache = dict()
    return wrapper_cache

@cache
@count_calls  # decorator that counts the invocations
def fibonacci(num):
    if num < 2:
        return num
    return fibonacci(num - 1) + fibonacci(num - 2)
Garbage collection in Python

CPython manages memory with a **reference counting** + a **mark&sweep** cycle collector scheme

- **Reference counting**: each object has a counter storing the number of references to it. When it becomes 0, memory can be reclaimed.
- **Pros**: simple implementation, memory is reclaimed as soon as possible, no need to freeze execution passing control to a garbage collector
- **Cons**: additional memory needed for each object; cyclic structures in garbage cannot be identified (thus the need of **mark&sweep**)


Handling reference counters

• Updating the refcount of an object has to be done atomically
• In case of multi-threading you need to synchronize all the times you modify refcounts, or else you can have wrong values
• Synchronization primitives are quite expensive on contemporary hardware
• Since almost every operation in CPython can cause a refcount to change somewhere, handling refcounts with some kind of synchronization would cause spending almost all the time on synchronization
The Global Interpreter Lock (GIL)

- The CPython interpreter assures that only one thread executes Python bytecode at a time, thanks to the **Global Interpreter Lock**
- The current thread must hold the **GIL** before it can safely access Python objects
- This simplifies the CPython implementation by making the object model (including critical built-in types such as `dict`) implicitly safe against concurrent access
- Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, **at the expense of much of the parallelism afforded by multi-processor machines.**
More on the GIL

• However the GIL can degrade performance even when it is not a bottleneck. The system call overhead is significant, especially on multicore hardware. Two threads calling a function may take twice as much time as a single thread calling the function twice.

• The GIL can cause I/O-bound threads to be scheduled ahead of CPU-bound threads. And it prevents signals from being delivered.

• Some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally-intensive tasks such as compression or hashing.

• Also, the GIL is always released when doing I/O.
Alternatives to the GIL?

• Past efforts to create a “free-threaded” interpreter (one which locks shared data at a much finer granularity) have not been successful because performance suffered in the common single-processor case.
• It is believed that overcoming this performance issue would make the implementation much more complicated and therefore costlier to maintain.
• Guido van Rossum has said he will reject any proposal in this direction that slows down single-threaded programs.
• **Jython** (in Java, -> 2015) and **IronPython** (on .NET) have no GIL and can fully exploit multiprocessor systems
• **PyPy** (Python in Python) currently has a GIL like CPython
• in **Cython** the GIL exists, but can be released temporarily using a "with" statement
Criticisms to Python: *scopes*

- In many languages, you have some nicely defined scopes (e.g., C++, Lisps). Scopes give you power to create readable and simple code. In Python, you only get TWO simple scopes - global, and function - and even handling these scopes is painful (keywords: global, nonlocal).

```python
def test():
    for a in range(5):
        b = a % 3
        print(b)
    print(b)

>>> test()
```

```python
def test(x):
    print(x)
    for x in range(5):
        print(x)
    print(x)

>>> test("Hello!")
```
Criticisms to Python: no closures

- No closures: because scoping is a foreign concept in Python, you don’t have proper closures.

```python
def counter_factory():
    counter = 0
    def counter_increaser():
        counter = counter + 1
        return counter
    return counter_increaser

>>> f = counter_factory()
>>> f()
Traceback (most recent call last):
  UnboundLocalError: local variable 'counter' referenced before assignment
```
Criticisms to Python: syntax of tuples

- Tuples are made by the commas, not by ( )
- With the exception of the empty tuple...

```python
>>> type((1,2,3))
<class 'tuple'>
>>> type(())
<class 'tuple'>
>>> type((1))
<class 'int'>
>>> type((1,))
<class 'tuple'>
```
Criticisms to Python: indentation

• Lack of brackets makes the syntax "weaker" than in other languages: accidental changes of indentation may change the semantics, leaving the program syntactically correct.

```python
def foo(x):
    if x == 0:
        bar()
        baz()
    else:
        qux(x)
    foo(x - 1)
```

• Mixed use of tabs and blanks may cause bugs almost impossible to detect
Criticisms to Python: indentation

• Lack of brackets makes it harder to refactor the code or insert new one (where should your if go?)
• "When I want to refactor a bulk of code in Python, I need to be very careful. Because if lost, I’m not sure what I’m editing belongs to which part of the code. Python depends on indentation, so if I have mistakenly removed some indentation, I totally have no idea whether the correct code should belong to that if clause or this while clause."
• Will Python change in the future?

>>> from __future__ import braces
>>> File "<stdin>", line 1
SyntaxError: not a chance
>>>
Exception Handling in Python (in 2 slides)

• Similar to Java
• Exceptions are Python objects
  – More specific kinds of errors are subclasses of the general `Error` class.
• You use the following forms to interact with them:
  – `try`
  – `except`
  – `else`
  – `finally`

for example...
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!"
    else:
        print("result is ", result
    finally:
        print("executing finally clause"

>>> divide(2, 1)
result is 2
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
execting finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str"