Announcement
- Exam is rescheduled back to Wednesday 19 dec, 8:45-11:30, in Atria
- See: http://www.few.vu.nl/onderwijs/roosters/Route-Atria.pdf

Outline
- Part I. Second look at Python
- Part II. Exam guidelines

Scope
- Python uses name spaces
- No prior definition needed before assignment
- The place of assignment determines the scope

LEGB scope rule
- When an unqualified name is used, Python searches 4 scopes:
  - L- local, E- enclosing def, G- global, and B- built-in and stops at the first place the name is found.
  - If it is not found, Python reports an error.
Nested scopes

- For example nested function definition
- Closure is a bundle referencing environment where the subroutine would execute if called now, together with a reference to the subroutine
- Originated in Lisp
- Possible in Ada, Modula 3, ML, Ruby

```python
>>> X = 88
>>> def func():
    X = 99;

>>> func()
>>> print X
88
>>> X=88
>>> def func():
    global X
    X = 89

>>> func()
>>> print X
89
```

Closure example

```python
>>> def f1():
    x = 88
    def f2():
        print x
        return x2

>>> action = f1()
>>> action()
88
```

First returns f2 as a closure that remembers the value of x from a scope that does not exist anymore, f1 already returned.

Second look at Python

- Scope
- Advanced function topics
- Object orientation
- Exceptions
- Python libraries

Advanced function topics

- Parameters
- Recursion
- Functional features
- List comprehension
- Generators

Passing arguments

- People don’t agree on how to call what happens in Python with passing parameters
- One way of saying is that Python uses “pass-by-assignment” [Lutz]
- Another way is to say that Python passes references to objects by value [Adam Weber].
Pass by assignment

When a function is defined, a, b are names.

```python
def func(a, b):
    print a, b
```

When the function is called, A, B are objects:

```python
func(A, B)
```

Names are assigned to the passed-in objects

Pass-by-assignment [Lutz] means that:

1. **Immutable arguments** act like passed by value (changes inside the functions do not affect the caller)
2. **Mutable arguments** act like passed by reference (changes inside the function affect the caller)

---

Returning multiple values

Python function can return more than one object by packaging them in a tuple.

```python
>>> def multiple(x, y):
...     return x, y
...>>> x=1
...>>> y=[1,2]
...>>> z,multiple(x,y)
...2, [1, 2]
...>>> x, y, z
...(2, [1, 2], 1)
```

---

Argument matching

- By position (by default)
- By keyword (match by name)
- Varargs (variable arguments list)
- Defaults (specify values for missing arguments)

---

Matching arguments

```python
>>> def f(a,b,c):
...     print a,b,c
...>>> f(1,2,'spam')  # Match by position
...1 2 spam
...>>> f('spam',2,1)  # Match by keyword (by name)
...1 2 spam
...>>> f(1,'spam',2)  # Mix positional and keywords
...1 spam
```

---

Why Keywords?

- Keywords make the calls more self-documenting
- Keywords can occur in conjunction with defaults

Ex: `func(name='Bob', age=40, job='clerk')`

```python
>>> def f(a=2,c=3):
...     print a, b, c
...>>> f(1,4)
...1 4 3
...>>> f(1,4,5)
...1 4 5
...>>> f(1,c=6)
...1 2 6
```
Keywords and defaults

```python
>>> def func(spam, eggs, toast=0, ham=0):
...     print(spam, eggs, toast, ham)  # first 2 required
... >>> func(1, 2)
1 2 0
>>> func(1, 2, 0, 0)
1 2 0
>>> func(1, 0, 0, 0)
1 0
>>> func(1, 0, 0, 0, 0)
1 0
>>> func(1, 0, 0, 0, 0, 0)
1 0
```

Any number of arguments: *

- Use *: Collects all unmatched parameters in a tuple that can be indexed, stepped through, etc.

```python
>>> def f(*args): print(args)
>>> f()()
>>> f(1)
(1,)
>>> f(1, 2, 3)
(1, 2, 3)
```

Any number of arguments: **

- Use **: Works only for keywords arguments, and collects them into a new dictionary.

```python
>>> def f(**kwargs): print(kwargs)
>>> f()
{}
>>> f(a=1, b=2)
{'a': 1, 'b': 2}
```

Recursion

```python
>>> def factorial(n):
...     if n == 1:
...         return 1
...     return n * factorial(n-1)
```

Functional features: lambda

- Lambda is an expression that returns an anonymous (unnamed) function to be called later.
- Lambda expression is a shorthand for def, to embed simple not reusable functions

```python
>>> def x(): return x*2
>>> x()  # Base case
0
>>> y=lambda x: x*2
>>> y()
0
>>> x=lambda x: x*2
```
Lambda can be used to code jump tables =
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)
4
8
16

Functional features: map
n Applies a function, passed as argument
over sequences:

```python
>>> counters=[1,2,3,4]
>>> def inc(x): return x+10

>>> map(inc, counters)
[11, 12, 13, 14]
>>> map((lambda x: x+10), counters)
[11, 12, 13, 14]
```

Functional features
n Reduce and operator (like foldr
and op in ML)

```python
>>> reduce((lambda x,y:x+y),[1,2,3,4])
10
>>> import operator
>>> reduce (operator.add,[1,2,3,4])
10
```

List comprehension
n List comprehensions collect the results of applying
an arbitrary expression on a sequence of values
and return them in a list

```python
[expr for var in seq]
```

n Can include/exclude elements in a list with
an if statement:

```python
[expr for var in seq if cond]
```

List comprehensions

```python
>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [x**2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
>>> zip(*[y for x in [0,1,2] for y in [100,200,300]])
>>> r=r

(100, 200, 300, 100, 200, 300)
```

Why not just use map or simply a for loop?
List comprehension is less to type and avoids for loops nesting
Map calls are twice as fast as for loops and list comprehensions are very slightly faster than maps (run at C speed inside the interpreter)
But for makes logic more explicit, so use maps and comprehension only when speed is important.

Generators
- Normal functions return a value and exit
- Generators are functions that may be resumed after they return a value
- The generator yields a value—suspending the function and sends a value back to caller, but retains enough state to allow function to resume from where it left off

How it works?
- When the function is called, the function is not executed but a generator object is generated, that supports the Python iterator protocols
- The generator object defines a next method, which returns the next item in the iteration or raises a special exception to end the iteration
- In the example, for calls the next method of the generator object.

Example
```python
>>> def genquares(n):
    for i in range(n):
        yield i**2

>>> for i in genquares(5):
    print(i, end=', ')
0:1:4:9:16:
```

Why not just use for, map or list comprehension?
```python
>>> x = [i**2 for i in range(5)]
>>> x
[0, 1, 4, 9, 16]
```
Generators are useful when lists are large or it takes much computation to produce each value—it spreads effort among for loop iterations.

Second look at Python

- Scope
- Advanced function topics
- Object orientation
- Exceptions
- Python libraries

OOP in Python

- OOP in Python is entirely optional (like in C++, not like in Java)
- We have seen that Python has objects, but
- To be qualified as truly OO, objects also have to participate in inheritance hierarchy

OOP model

- Python OOP model has 2 kind of objects: classes and instances.
- Class objects provide behavior and are instance objects factories.
- Instance objects are concrete items, real objects your program processes
- Classes come from statements, instances come from calls

Classes and instances

```python
class C2: ... # make class objects
class C3: ...
class C1(C2, C3) # C1 inherits C2 and C3
I1 = C1() # make instance objects
I2 = C1()
```

Inheritance

- Classes can be customized by inheritance
- Instance objects inherit from class objects
- Class objects can inherit from other class objects (called superclasses)
- Classes and instances are members of a big inheritance tree
- Python support multiple inheritance
OOP model

- All OOP story is about object.attribute
- Each object.attribute invokes in Python a search for attribute in the inheritance tree.
- Python first checks in the instance object, then its class, then all superclasses of its class, from bottom to top and from left to right.

Class Example

```python
>>> class FirstClass:
    def setdata(self, value):
        self.data = value
    def display(self):
        print(self.data)

>>> a = FirstClass()
>>> b = FirstClass()
>>> a.setdata('king Arthur')
>>> b.setdata(3.14)

>>> a.display()  # 'king Arthur'
>>> b.display()  # 3.14
```

Classes

- A class has fields and methods = attributes.
- Class methods are functions, but
- Their first parameter is always self (a kind of this from C++ and Java)
- When a instance method is called, like x.setdata (3.14), Python converts this automatically in a class method call FirstClass (x, 3.14). So self will refer to instance x.

__init__ method

__init__ method has a special significance for a class
Useful for initialization, is like a constructor in C++ or Java

```python
>>> class Person:
    def __init__(self, name):
        self.name = name
    def sayHello(self):
        print('Hello, my name is', self.name)

>>> p = Person('Python')
>>> p.sayHello()
Hello, my name is Python
```

Dynamic method binding

```python
>>> class Employee:
    def computeSalary(self): return self.salary * 2

>>> class Manager(Employee):
    def __init__(self):
        self.salary = 1000

>>> bob = Employee()
>>> bobi = Manager()
>>> print(bob.computeSalary())
1000
```
Things can go wrong

```python
>>> x=5
Traceback (most recent call last):
  File "<ipython-input-1-778405fb540e>", line 1, in <module>
    x=x
NameError: name 'x' is not defined
>>> 1/0
Traceback (most recent call last):
  File "<ipython-input-1-778405fb540e>", line 1, in <module>
    1/0
ZeroDivisionError: integer division or modulo by zero
>>> 'spam' 'eggs'
Traceback (most recent call last):
  File "<ipython-input-1-778405fb540e>", line 1, in <module>
    TypeError: cannot concatenate 'str' and 'int' objects
```

Exceptions

Exception — event that modifies the flow of control
Automatically raised by Python when errors occur
Can be intercepted or not by the user code
Can be also raised by the user code
A mechanism similar to Java

Example

Case 1: Everything works fine
```python
>>> def fetcher(obj, index):
    return obj[index]
>>> x = 'spam'
>>> fetcher(x, 3)
'p'
```

Case 2: An exception of type `IndexError` is raised, but the user code does nothing about it. Default exception behaviour.
```python
>>> fetcher(x, 4)
Traceback (most recent call last):
  File "<ipython-input-1-778405fb540e>", line 1, in <module>
    fetcher(x, 4)
IndexError: list index out of range
```

Exception handler

Case 3: The same exception is raised, but the user code catches it, using a `try/except` block (exception handler)
```python
>>> try:
    fetcher(x, 4)
except IndexError:
    print 'got an exception'
got an exception
```

Or even better, with `recover`
```python
>>> def fetcher(obj, index):
    try:
        return obj[index]
    except IndexError:
        print 'got exception'
        print 'continuing'
>>> fetcher(x, 4)
got exception continuing
```

Another example
```python
>>> fsock = open('./notthere.txt', 'r')
Traceback (most recent call last):
  File "<ipython-input-1-778405fb540e>", line 1, in <module>
    fsock = open('./notthere.txt', 'r')
IOError: [Errno 2] No such file or directory: '/notthere.txt'
>>> not found()
```
Example

```python
>>> for x in range(-3,3):
    print x,
try:
    print 1.0/x
except ZeroDivisionError:
    print "Error"
```

-3 -0.33333333333333333
-2 -0.5
-1 -1.0
0 Error
1 1.0
2 0.5

User-made exceptions: Raise

```python
>>> try:
    raise ZeroDivisionError
except ZeroDivisionError:
    print 'we got division by zero'
```

we got division by zero

Catching everything

```python
>>> try:
    5/0
except Exception:
    print "got a serious error!"
```

not a serious error!

Exceptions hierarchy

- Exceptions are class objects in Python.
- The root class for all exceptions is the class Exception
- See http://www.python.org/doc/current/ for a full listing of Python exceptions and their hierarchy
- Knowing the hierarchy can help the user in catching a group of exceptions

Finally

- Guarantees termination actions (like closing files) regardless of the exceptions that may occur in the try block
Second look at Python

- Scope
- Advanced function topics
- Object orientation
- Exceptions
- Python libraries

Python libraries (modules)
- Python comes with powerful built-in function and modules (libraries)
- It is not necessary to reinvent the wheel
- Examples: String and re modules

String module
- Has strong text processing functions
- For example maketrans function to translate a few characters at once

```
>>> import string
>>> conversion = string.maketrans("_", "_", ")
>>> import string
>>> conversion = "This is black box white box game"
>>> import re
>>> re_string = re.escape(conversion)
'\This\ is\ block\ box\ white\ box\ game\'
```

Example
```
>>> s = '100 NORTH MAIN ROAD'
>>> s.replace('ROAD', 'RD.')
'100 NORTH RD.'
>>> s = '100 NORTH BROAD ROAD'
>>> s.replace ('ROAD', 'RD.')
'100 NORTH RD.'
```
Flying with Python

Bill Venners: Speaking of spacecraft, would you be comfortable enough with the robustness of Python systems to fly on an airplane in which all the control software was written in Python?

Guido van Rossum: That depends much more on the attitude of the design team that built it than on the language the team chose. There are situations where doing part of the software in Python makes much more sense than doing it in any other language, even if it must have the reliability requirements of a spacecraft or air traffic control. Why?

Guido van Rossum: You'll never get all the bugs out. Making the code easier to read and write, and more transparent to the team of human readers who will review the source code, may be much more valuable than the narrow-focused type checking that some other compiler offers. There have been reported anecdotes about spacecraft or aircraft crashing because of type-related software bugs, where the compilers weren't enough to save you from the problems.
Exam guidelines

- The question will try to cover all the studied stuff, including your assignment
- All studied languages will be represented
- Final grade = (Q1+Q2+...) / 10
- Sample exam and solutions can be found on the PPL website
- You have to know everything presented during lectures, but more precisely ...

<table>
<thead>
<tr>
<th>Lecture</th>
<th>Conceptual terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>History of PL</td>
</tr>
<tr>
<td>2</td>
<td>Syntax, semantics, grammars, precedence, associativity, ambiguity</td>
</tr>
<tr>
<td>3</td>
<td>Classical sequence, binding times</td>
</tr>
<tr>
<td>4</td>
<td>Primitive types, constructed types, enumerations, tuples, type inference, annotations, type checking, type equivalence</td>
</tr>
<tr>
<td>5</td>
<td>Inheritance, overriding, polymorphism, ML: Anonymous function, pattern matching, high order functions carrying</td>
</tr>
<tr>
<td>6</td>
<td>Scope, labeled namespace, primitive namespace, static scoping, dynamic scoping, activation records, static allocation, stack allocation, heap allocation, garbage collection</td>
</tr>
<tr>
<td>7</td>
<td>Parameter correspondence, parameter passing, throwing/catching exceptions in Java, finally, resource handling procedures</td>
</tr>
<tr>
<td>8</td>
<td>Logic: terms, atoms, facts and rules, L calculus, append, member, flexible vs. inflexible predicates, negation and failure, unification, backtracking, cut</td>
</tr>
<tr>
<td>9-10</td>
<td>Objects, fields, methods, inheritance, polymorphism, static/dynamic method binding, Python: tuples, strings, lists, dictionaries, functions, list comprehension, OOP, exceptions</td>
</tr>
</tbody>
</table>