Principles of programming languages

Lecture 10

Natalia Silvis-Cividjian
e-mail: nsilvis@few.vu.nl

vrije Universiteit *amsterdam*

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

Second look at Python

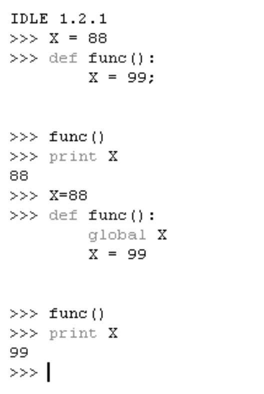
➡
- Scope
- Advanced function topics
- Object orientation
- Exceptions
- Python libraries

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.

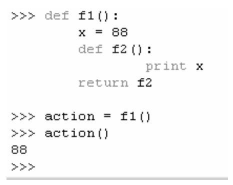


```
IDLE 1.2.1
>>> X = 88
>>> def func():
        X = 99;


>>> func()
>>> print X
88
>>> X=88
>>> def func():
        global X
        X = 99


>>> func()
>>> print X
99
>>> |
```

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

Closure example



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

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

f1 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**.

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

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

`func(A,B)`

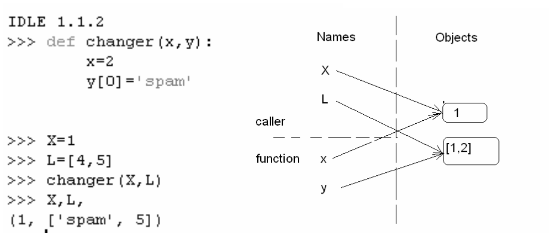
Names are assigned to the passed-in objects

Pass-by-assignment [Lutz] means that:

§Immutable arguments act like passed by value (changes inside the functions do not affect the caller)

§Mutable arguments act like passed by reference (changes inside the function affect the caller)

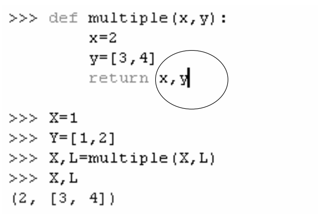
Pass by assignment



```
IDLE 1.1.2
>>> def changer(x,y):
        x=2
        y[0]='spam'


>>> X=1
>>> L=[4,5]
>>> changer(X,L)
>>> X,L,
(1, ['spam', 5])
```

Returning multiple values

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



```
>>> def multiple(x,y):
        x=2
        y=[3,4]
        return x,y

>>> X=1
>>> Y=[1,2]
>>> X,L=multiple(X,L)
>>> X,L
(2, [3, 4])
```

Argument matching

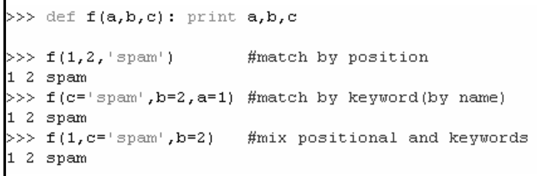
n By position (by default)

n By keyword (match by name)

n Varargs (variable arguments list)

n Defaults (specify values for missing arguments)

Matching arguments



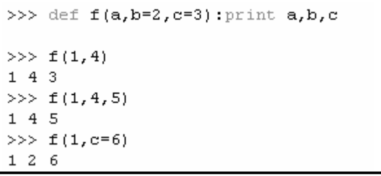
```
>>> def f(a,b,c): print a,b,c

>>> f(1,2,'spam')        #match by position
1 2 spam
>>> f(c='spam',b=2,a=1)  #match by keyword(by name)
1 2 spam
>>> f(1,c='spam',b=2)    #mix positional and keywords
1 2 spam
```

Why Keywords ?

n Keywords make the calls more self-documenting

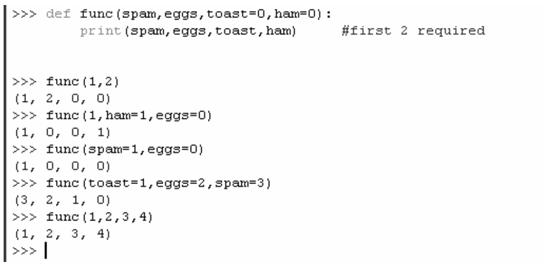
`Ex: func(name=‘Bob’,age=40,job=‘ clerk’)`

n Keywords can occur in conjunction with defaults



```
>>> def f(a,b=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
```

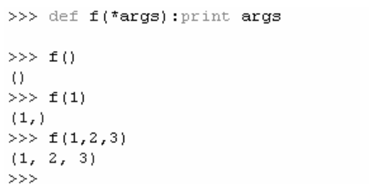
Keywords and defaults



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

>>> func(1,2)
(1, 2, 0, 0)
>>> func(1,ham=1,eggs=0)
(1, 0, 0, 1)
>>> func(spam=1,eggs=0)
(1, 0, 0, 0)
>>> func(toast=1,eggs=2,spam=3)
(3, 2, 1, 0)
>>> func(1,2,3,4)
(1, 2, 3, 4)
>>>
```

Any number of arguments: *

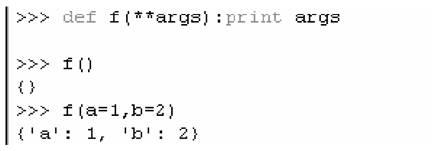
- Use *. Collects all unmatched parameters in a tuple that can be indexed, stepped through, etc.



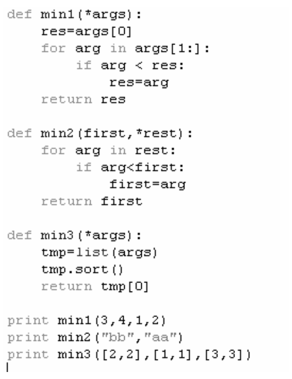
```
>>> def f(*args):print args

>>> 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.

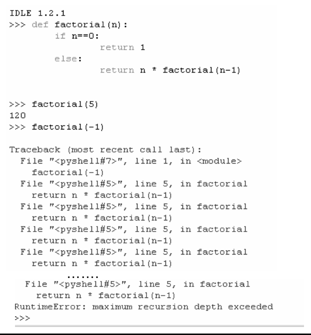


```
>>> def f(**args):print args

>>> f()
{}
>>> f(a=1,b=2)
{'a': 1, 'b': 2}
```



```
def min1(*args):
    res=args[0]
    for arg in args[1:]:
        if arg < res:
            res=arg
    return res

def min2(first,*rest):
    for arg in rest:
        if arg<first:
            first=arg
    return first

def min3(*args):
    tmp=list(args)
    tmp.sort()
    return tmp[0]

print min1(3,4,1,2)
print min2("bb","aa")
print min3([2,2],[1,1],[3,3])
```

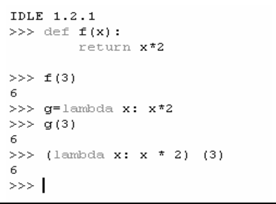
Recursion



```
IDLE 1.2.1
>>> def factorial(n):
        if n==0:
                return 1
        else:
                return n * factorial(n-1)


>>> factorial(5)
120
>>> factorial(-1)

Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    factorial(-1)
  File "<pyshell#5>", line 5, in factorial
    return n * factorial(n-1)
  File "<pyshell#5>", line 5, in factorial
    return n * factorial(n-1)
  File "<pyshell#5>", line 5, in factorial
    return n * factorial(n-1)
  File "<pyshell#5>", line 5, in factorial
    return n * factorial(n-1)
        .......
  File "<pyshell#5>", line 5, in factorial
    return n * factorial(n-1)
RuntimeError: maximum recursion depth exceeded
>>>
```

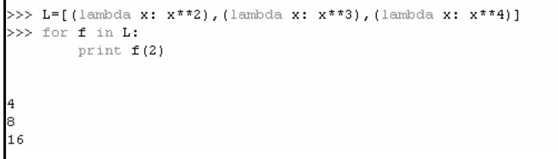
Functional features: lambda

- 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



```
IDLE 1.2.1
>>> def f(x):
        return x*2

>>> f(3)
6
>>> g=lambda x: x*2
>>> g(3)
6
>>> (lambda x: x * 2) (3)
6
>>>
```

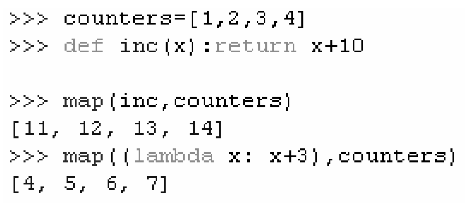
- Lambda can be used to code jump tables =
lists or dictionaries of actions to be performed on deman



```
>>> 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

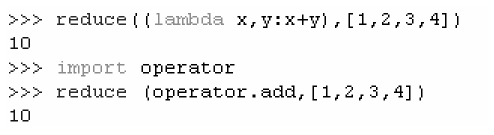
- Applies a function, passed as argument over sequencies:



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

>>> map(inc,counters)
[11, 12, 13, 14]
>>> map((lambda x: x+3),counters)
[4, 5, 6, 7]
```

Functional features

- `Reduce` and `operator` (like `foldr` and op in ML)

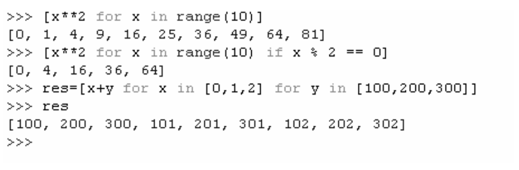


```
>>> reduce((lambda x,y:x+y),[1,2,3,4])
10
>>> import operator
>>> reduce (operator.add,[1,2,3,4])
10
```

List comprehension

- List comprehensions collect the results of applying an arbitrary expression on a sequence of values and return them in a list
  `[expr for var in seq]`
- Can include/exclude elements in a list with an if statement:
  `[expr for var in seq if cond]`

List comprehensions



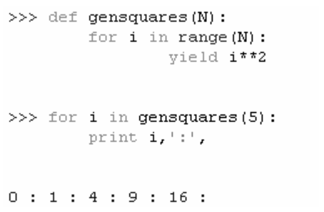
```
>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [x**2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
>>> res=[x+y for x in [0,1,2] for y in [100,200,300]]
>>> res
[100, 200, 300, 101, 201, 301, 102, 202, 302]
>>>
```

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=suspends the function and sends a value back to caller, but retains enough state to allow function to resume from where it left off

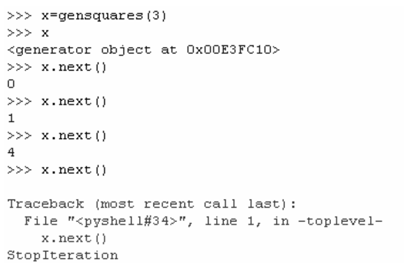


```
>>> def gensquares(N):
        for i in range(N):
                yield i**2


>>> for i in gensquares(5):
        print i,':',


0 : 1 : 4 : 9 : 16 :
```

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



```
>>> x=gensquares(3)
>>> x
<generator object at 0x00E3FC10>
>>> x.next()
0
>>> x.next()
1
>>> x.next()
4
>>> x.next()

Traceback (most recent call last):
  File "<pyshell#34>", line 1, in -toplevel-
    x.next()
StopIteration
```

Why not just use for, map or list comprehension?

- 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

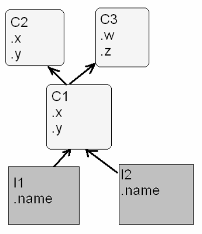
OOP in Python

- 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



```
class C2:... #make class objects
class C3:...
class C1(C2,C3) #C1 inherits C2
                #and C3

I1=C1()  #make instance objects
I2=C1()
```

Inheritance

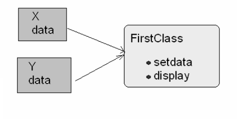
- Classes can be customized by inheritance
- Instance objects inherit from class objects
- Class objects can inherit from other class objects (called superclasses)
- Classes and instances are members of a big inheritance tree
- Python support multiple inheritance

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

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


>>> x=FirstClass()      #make 2 instances
>>> y=FirstClass()
>>> x.setdata("king Arthur")
>>> y.setdata(3.14)
>>> x.display()
king Arthur
>>> y.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

```
>>> 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

```
>>> class Employee:
        def computeSalary(self): return (200)


>>> class Engineer(Employee):   #a subclass
        def computeSalary(self): return (500)


>>> bob = Employee()
>>> sheila=Engineer()
>>> company = [bob, sheila]
>>> for emp in company:
        print emp.computeSalary()


200
500
>>> |
```

Second look at Python

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

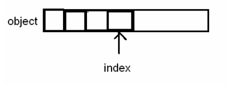
Things can go wrong

```
>>> x==5

Traceback (most recent call last):
  File "<pyshell#1>", line 1, in -toplevel-
    x==5
NameError: name 'x' is not defined
>>> 1/0

Traceback (most recent call last):
  File "<pyshell#2>", line 1, in -toplevel-
    1/0
ZeroDivisionError: integer division or modulo by zero
>>> 'Spam'+5

Traceback (most recent call last):
  File "<pyshell#3>", line 1, in -toplevel-
    'Spam'+5
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

```
>>> def fetcher(obj,index):
        return obj[index]

>>> x='spam'
>>> fetcher(x,3)
'm'
```

Case 2. An exception of type `IndexError` is raised, but the user code does nothing about it. Default exception behaviour.

Stack trace

```
>>> fetcher(x,4)

Traceback (most recent call last):
  File "<pyshell#68>", line 1, in -toplevel-
    fetcher(x,4)
  File "<pyshell#65>", line 2, in fetcher
    return obj[index]
IndexError: string 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)

```
>>> try:
        fetcher(x,4)
except:
        print "got an exception"


got an exception
```

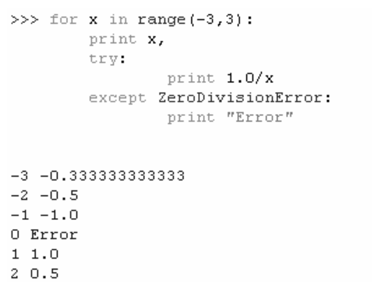
Or even better, with recover

```
>>> def catcher():
        try:
                fetcher('spam',4)

        except IndexError:
                print 'got exception'
        print 'continuing'


>>> catcher()
got exception
continuing
```

Another example

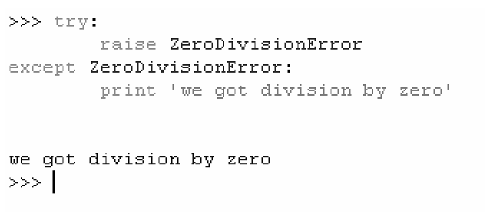
```
>>> fsock = open ("/nothere","r")

Traceback (most recent call last):
  File "<pyshell#54>", line 1, in <module>
    fsock = open ("/nothere","r")
IOError: [Errno 2] No such file or directory: '/nothere'
>>> def catcher():
        try:
                fsock = open ("/notthere","r")
        except IOError:
                print "The file does not exist, exiting"
        print "continuing"


>>> catcher()
The file does not exist, exiting
continuing
>>> |
```

Example



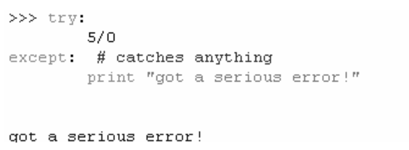
```
>>> for x in range(-3,3):
	print x,
	try:
		print 1.0/x
	except ZeroDivisionError:
		print "Error"


-3 -0.333333333333
-2 -0.5
-1 -1.0
0 Error
1 1.0
2 0.5
```

User-made exceptions: Raise



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


we got division by zero
>>> |
```

Catching everything

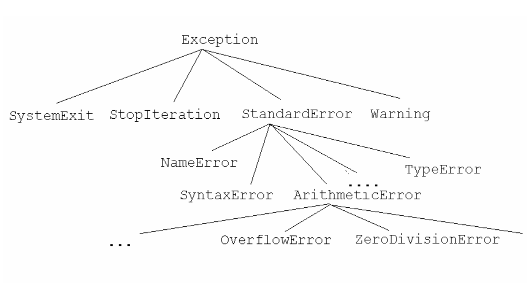


```
>>> try:
	5/0
except:  # catches anything
	print "got a serious error!"


got a serious error!
```

Exceptions hierarchy

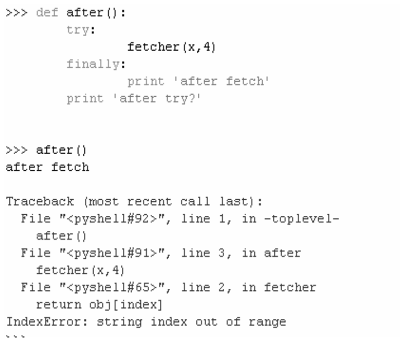
- Exceptions are class objects in Python.
- The root class for all exceptions is the class `Exception`
- See http://www.python.org/doc/current/ for a full listing of Python exceptions and their hierarchy
- Knowing the hierarchy can help the user in catching a group of exceptions

Exceptions hierarchy



Finally

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



```
>>> def after():
        try:
                fetcher(x,4)
        finally:
                print 'after fetch'
        print 'after try?'


>>> after()
after fetch

Traceback (most recent call last):
  File "<pyshell#92>", line 1, in -toplevel-
    after()
  File "<pyshell#91>", line 3, in after
    fetcher(x,4)
  File "<pyshell#65>", line 2, in fetcher
    return obj[index]
IndexError: string index out of range
>>>
```

Second look at Python

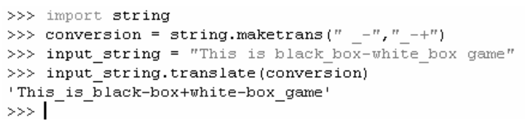
- Scope
- Advanced function topics
- Object orientation
- Exceptions
- Python libraries

Python libraries (modules)

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

`String` module

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

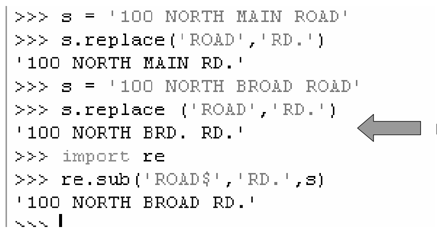


```
>>> import string
>>> conversion = string.maketrans(" _-","_-+")
>>> input_string = "This is black_box-white_box game"
>>> input_string.translate(conversion)
'This_is_black-box+white-box_game'
>>> 
```

`re` module

- If string module methods are not enough, `re` module can help.
- Python regular expressions (re) engine is a specialized string-processing tool
- re can compactly express complicated matching rules necessary for text processing
- re is a highly-specialized tiny programming language embedded inside Python

Example



```
>>> s = '100 NORTH MAIN ROAD'
>>> s.replace('ROAD','RD.')
'100 NORTH MAIN RD.'
>>> s = '100 NORTH BROAD ROAD'
>>> s.replace ('ROAD','RD.')
'100 NORTH BRD. RD.'
>>> import re
>>> re.sub('ROAD$','RD.',s)
'100 NORTH BROAD RD.'
>>> 
```

problem

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.

Outline

Part I. Second look at Python

➡ **Part. II. Exam guidelines**

Finally, what did you like/not like/remember from this PPL course?

Final conclusion

- The best programming language does not exist
- A lot of trade-offs in PL design and implementation
- A lot of PL groups exist and fight
- Influences on programming languages evolution
- You know now how to evaluate a new PL

Assessment

- Final grade = 0.3 * assignment + 0.7 * exam
- Both components have to be > 5.5

The exam

- The exam is written and closed-book.
- Consists of 2 types of questions:
  - Define and explain theoretical PL concepts
  - Explain how a code snippet works and what output is generated under given circumstances.

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 …

Lecture	Conceptual terms
1	History of PL
2	Syntax,semantics Grammars, precedence, associativity, ambiguity
3	Classical sequence, binding times
4	Primitive types, constructed types Enumerations, tuples Type inference, annotations, type checking, type equivalence
5	Polymorphism, overloading, coercion ML: Anonymous function, pattern matching, high order functions currying
6	Scope, labeled namespace, primitive namespace Static scoping, dynamic scoping Activation records, static allocation, stack allocation, heap allocation, garbage collection
7	Parameter correspondence, parameter passing Throwing/catching exceptions in Java, finally, error handling procedures,
8	Prolog: terms, atoms, facts and rules Lists, append, member, flexible vs. inflexible predicates, negation and failure, unification, backtracking, cut
9+10	Objects, fields, methods Inheritance, polymorphism, static/dynamic method binding Python: tuples, strings, lists, dictionaries, functions, list comprehension, OOP, exceptions