Introduction to Python

Alastair Burt   Andreas Eisele   Christian Federmann
Torsten Marek   Ulrich Schäfer

Universität des Saarlandes

October 6th, 2009
Outline

Today’s Topics:

1. More Examples
2. Cool Stuff
3. Text Processing
4. Persistence
5. Modules
Part I

More Examples
More exception handling

The ideal world

Consider things like database connections or files: they are opened, some data is read or written and at some point they should be closed.
More exception handling

The ideal world

Consider things like database connections or files: they are opened, some data is read or written and at some point they should be closed.

Programmer’s reality

In the real world, we often encounter problems or program crashes. This is bad as our resource handle(s) remain ”opened” and waste memory, etc.
Exception syntax

Solution

As we have seen before, exceptions can be used to report and react on errors. The full syntax looks like this:

```python
try:
    open file/database/whatsoever
except:
    handle any raised exception here
finally:
    close handle
```
Exception syntax

Solution

As we have seen before, exceptions can be used to report and react on errors. The full syntax looks like this:

```
try:
    open file/database/whatever
except:
    handle any raised exception here
finally:
    close handle
```

finally is always executed

The finally block will always be executed so we can be sure that our resources are released again.
**finally is a ”cleanup” handler**

If a `finally` block is present, any exception that is not handled will be saved temporarily. After the try and except blocks have been processed, the code inside the `finally` block gets executed before the saved exceptions are raised again.
finally is a "cleanup" handler

If a finally block is present, any exception that is not handled will be saved temporarily. After the try and except blocks have been processed, the code inside the finally block gets executed before the saved exceptions are raised again.

Restriction

The exception information is not available inside finally block, use exception handlers beforehand.
Recap: Searching

Searching Functions

- `match` only finds a match at the beginning of the string
- `search` scans the whole string
- `findall` returns a list of all matches in a string
- `finditer` returns an iterator of all matches
Recap: Searching

Searching Functions

- `match` only finds a match at the beginning of the string
- `search` scans the whole string
- `.findall` returns a list all matches in a string
- `finditer` returns an iterator of all matches

Other Functions

- `split` uses the regex as a splitter
- `sub`, `subn` substitution patterns
**match vs. search**

`regexp.match(str)` checks if the regexp matches the beginning of the given string `str`, otherwise `None` is returned. Example:

```python
>>> regexp = re.compile("auto")
>>> regexp.match("autobot")
<_sre.SRE_Match object at 0x50e90>  

>>> regexp.match("decepticon") == None
True
```
**match vs. search**

`regexp.search(str)` checks if the regexp matches any part of the given string `str`. Again, `None` is returned when the regexp cannot be found:

```python
>>> regexp.search("fully automatic")
<_sre.SRE_Match object at 0x50e90 >

>>> regexp.search("decepticon") == None
True
```
findall vs. finditer

regexp.findall(str) returns a list of all matches of the given string str. Note that this can be very slow for large strings:

```python
>>> regexp = re.compile("ab")
>>> regexp.findall("abracadabra")
['ab', 'ab']
```
findall vs. finditer

regexp.finditer(str) returns an iterator object that will (step by step) return all matches of the given string str. You can use a for loop to access the matches:

```python
>>> for m in regexp.finditer("abracadabra"):  
...     print m 
...
<_sre.SRE_Match object at 0x50e90> 
<_sre.SRE_Match object at 0x79720> 
```
split

regexp.split(str) uses the given regular expression to split the given string str into smaller chunks and returns those in a list:

```python
>>> regexp = re.compile("ab.")
>>> regexp.split("abracadabra")
['', 'acad', 'a']
```
More regular expressions

**split**

`regexp.split(str)` uses the given regular expression to split the given string `str` into smaller chunks and returns those in a list:

```python
grep = re.compile("ab.")
grep.split("abracadabra")
['', 'acad', 'a']
```

**split also available as string method!**

For simple split expressions it is more efficient to use the built-in `split` method for strings.
More regular expressions

**sub vs. subn**

`regexp.sub(replace, str)` replaces any matching substring inside the given string `str` with the value in `replace`. The `subn` method returns a tuple containing the resulting string and the number of replacements made:

```python
>>> regexp.sub("foo", "abracadabra")
'fooacadfooa'

>>> regexp.subn("foo", "abracadabra")
('fooacadfooa', 2)
```
Part II

Cool Stuff
Cool Stuff

Builtin Functions

Python has some nice builtin methods to work with iterable containers such as lists and tuples:

- `all` returns true if all elements in the container evaluate to True
- `any` returns true if at least one element in the container evaluates to True
- `enumerate` returns an iterator object which will return tuples containing a count (from zero) and the element from the iterable container
Built-in Functions, cont’d

Python has some nice built-in methods to work with iterable containers such as lists and tuples:

- **filter** return those values from a list for which the given condition holds
- **map** scans through the list and calls a function on each element
- **reduce** allows to compute a "reduced" value from the iterable elements
Built-in Functions, cont’d

Python has some nice built-in methods to work with iterable containers such as lists and tuples:

- `max` computes the largest element from a container
- `min` computes the smallest element from a container
- `sum` returns the sum of all list elements
- `zip` creates tuples out of multiple containers
Built-in Functions, cont’d

Python has some nice built-in methods to work with iterable containers such as lists and tuples:

- `max` computes the largest element from a container
- `min` computes the smallest element from a container
- `sum` returns the sum of all list elements
- `zip` creates tuples out of multiple containers

Elegant coding

This allows for very elegant coding, instead of using loops just work on iterables.
List Comprehension

A list comprehension is a syntactic construct available in some programming languages for creating a list based on existing lists. – Wikipedia, List Comprehension
List Comprehension

A list comprehension is a syntactic construct available in some programming languages for creating a list based on existing lists. – Wikipedia, List Comprehension

But I could use a for loop to do that, or?
Sure, but... that’s not Pythonic ;)
List Comprehension

This allows to compute new lists from existing lists without having to create multiple lines of code. Think of reversing dictionary items which are tuples:

```python
>>> freqs = {"good": 1, "bad": 1, "news": 2}
>>> freqs.items()
[('news', 2), ('bad', 1), ('good', 1)]
```
List Comprehension

This allows to compute new lists from existing lists without having to create multiple lines of code. Think of reversing dictionary items which are tuples:

```python
>>> freqs = {"good": 1, "bad": 1, "news": 2}
>>> freqs.items()
[('news', 2), ('bad', 1), ('good', 1)]
```

Desired result

```
[(2, 'news'), (1, 'bad'), (1, 'good')]
```
Possible Solution

We can iterate over all keys and create new tuples that get inserted into a new list:

```python
>>> result = []
>>> for key in freqs.keys():
...    new_tuple = (freqs[key], key)
...    result.append(new_tuple)
...
>>> result
[(2, 'news'), (1, 'bad'), (1, 'good')]
```
List comprehension

Possible Solution

We can iterate over all keys and create new tuples that get inserted into a new list:

```python
>>> result = []
>>> for key in freqs.keys():
...    new_tuple = (freqs[key], key)
...    result.append(new_tuple)
...
>>> result
[(2, 'news'), (1, 'bad'), (1, 'good')]
```

Think what happens for multiple for loops

Certainly not that elegant code...
Better Solution

Using list comprehension this whole mess shrinks to the following one-liner:

```python
>>> [(x[1], x[0]) for x in freqs.items()]
[(2, 'news'), (1, 'bad'), (1, 'good')]
```
Better Solution

Using list comprehension this whole mess shrinks to the following one-liner:

```python
>>> [(x[1], x[0]) for x in freqs.items()]
[(2, 'news'), (1, 'bad'), (1, 'good')]
```

We can combine several list comprehensions...

```python
>>> a = ['john', 'paul', 'mary']
>>> b = ['smith', 'meyer']
>>> ['%s %s' % (x, y) for x in a for y in b]
['john smith', 'john meyer', 'paul smith', ...
 'paul meyer', 'mary smith', 'mary meyer']
```
Remember the `try... finally` blocks? Python knows something called the `with` statement that allows to replace these elegantly...
With Statement

Remember the try... finally blocks? Python knows something called the with statement that allows to replace these elegantly...

Well, why would I do that?

- makes your code more readable
- helps to focus on the algorithm rather than syntax
We need a class

OK, we don’t know classes yet, so don’t get confused. For now, a class is just some “thing” that opens a file and closes it when we are done.

```python
>>> class gatekeeper:
...     def __enter__(self):
...         open file
...         return thing
...     def __exit__(self, type, value, tb):
...         close file

>>> with gatekeeper() as thing:
...     do something with the file
```
With Statement

Batteries included

Luckily, this is built-in for files! Hence we can avoid the whole exception handler code and "just work" with our files. Nice.

```python
>>> with open("some.txt", "r") as f:
...     for line in f:
...         print line
```
With Statement

Batteries included

Luckily, this is built-in for files! Hence we can avoid the whole exception handler code and "just work" with our files. Nice.

```python
>>> with open("some.txt", "r") as f:
...     for line in f:
...         print line
```

exit is always called...

Python will always execute the exit method, **no matter what happens** in the "do something" block! Yes, that means **even** if exceptions are raised. Also, exit has access to the exception information and can work with it.
Lambda Functions

Functions are first-class objects

Python allows to construct functions at runtime and to store them in variables and data structures. These functions are called `lambda` functions and can be used anywhere a function is required. They can also be passed as arguments to other functions.

A Lambda Example

The following example defines an inline function that increments a given int by a value of 10 and binds it to a variable named `func`:

```
func = lambda x: x + 10
```
Lambda Functions

Functions are first-class objects

Python allows to construct functions at runtime and to store them in variables and data structures. These functions are called `lambda` functions and can be used anywhere a function is required. They can also be passed as arguments to other functions.

A Lambda Example

The following example defines an `inline` function that increments a given `int` by a value of 10 and binds it to a variable named `func`:

```python
func = lambda x: x + 10
```
Lambda functions instead of ”real” functions

We can now rewrite our previous map example using our previously defined lambda function:

```python
>>> a = [1, 2, 3, 4, 5]
>>> map(lambda x: x ** 2, a)
[1, 4, 9, 16, 25]
```
Lambda functions instead of "real" functions

We can now rewrite our previous `map` example using our previously defined lambda function:

```python
>>> a = [1, 2, 3, 4, 5]
```

```python
>>> map(func, a)
[11, 12, 13, 14, 15]
```

Anonymous lambda functions

We don’t even have to introduce a variable that binds our lambda function, we can simply provide the code instead:

```python
>>> map(lambda x: x ** 2, a)
[1, 4, 9, 16, 25]
```
Brief: Generator Functions

Definition

In computer science, a generator is a special routine that can be used to control the iteration behaviour of a loop. (...) In short, a generator looks like a function but behaves like an iterator. – Wikipedia
Brief: Generator Functions

Iterators with knowledge

```python
>>> def fib():
...     a = 1
...     b = 1
...     while True:
...         yield a
...     a = a + b
...     b = a - b

>>> f = fib()
>>> f.next()  # returns 1
>>> f.next()  # returns 2
>>> f.next()  # returns 3
>>> f.next()  # returns 5
```
Part III

Lightweight Persistence
Motivation

Often, it is desired to store data structures between several invocations of a program without much further work.
Motivation

Often, it is desired to store data structures between several invocations of a program without much further work.

Possible Solutions

- ad-hoc file formats - needs parsing and writing code
- embedded databases - often overkill, needs SQL
- serialization!
Serialization: pickle

Contents

- reads and writes arbitrary (!) Python data structures
- supports exchange between:
  - invocations of a program
  - users on the same machine
  - interpreters
  - machines
Serialization: *pickle*

Contents

- reads and writes arbitrary (!) Python data structures
- supports exchange between:
  - invocations of a program
  - users on the same machine
  - interpreters
  - machines

Managing Pickles

The *shelve* module supports storing and reading any number of pickles from an embedded database.
A Pickle Example

A Short Example

We can "pickle" a list as follows:

```python
>>> import pickle
>>> data = pickle.dumps(['john', 'paul'])
>>> data
'(lp0\nS 'john ' np1\naS 'paul ' np2\na .'

>>> ['john', 'paul'] == pickle.loads(data)
True
```
A Pickle Example

A Short Example

We can "pickle" a list as follows:

```python
>>> import pickle
>>> data = pickle.dumps(['john', 'paul'])
>>> data
'(lp0\nS'john'\np1\naS'paul'\np2\na.)

>>> ['john', 'paul'] == pickle.loads(data)
True
```

More Batteries Included

You can get that for free and switch to something more elaborate once you have time to build that...
Part IV

Text Processing
Before We Begin

Beauty vs. Speed

- focus in clarity and expression
- speed is a minor issue for now
- underlying methods are supposed to be "perfect" performance-wise
A Closer Look at Strings

Why Aren’t Strings Just Lists of Characters?

- performance
- C heritage
- strings need a lot of special methods / algorithms
- performance
A Closer Look at Strings

Why Aren’t Strings Just Lists of Characters?

- performance
- C heritage
- strings need a lot of special methods / algorithms
- performance

Strings as Sequences

1. Python strings are ordinary sequences.
2. Non-modifying functions that work on lists will normally work on strings.
The Problem

Definition

A palindrome is a word, phrase, number or other sequence of units that has the property of reading the same in either direction (the adjustment of punctuation and spaces between words is generally permitted) – Wikipedia, Palindrome
The Problem

Definition

A palindrome is a word, phrase, number or other sequence of units that has the property of reading the same in either direction (the adjustment of punctuation and spaces between words is generally permitted) – Wikipedia, Palindrome

Palindrome Checker

How?
The Palindrome Checker

A Simple Solution

def is_palindrome(a):
    return list(a) == list(reversed(a))
The Palindrome Checker

A Simple Solution

```python
def is_palindrome(a):
    return list(a) == list(reversed(a))
```

reversed

`reversed` takes a sequence and returns an `iterator` that yields all values from the input sequence in reverse order.
What Is An Iterator?

A First Glance at Laziness

- "lazy" sequences
- useful for iteration
- piece-wise retrieval of elements
- trades time for space
How To Use Iterators

Using Iterators

generally: iterators are sequences
for loops: no change
as lists: explicit conversion
How To Use Iterators

Using Iterators

- generally: iterators are sequences
- for loops: no change
- as lists: explicit conversion

Iterators as Sequences

1. Python iterators can be used as "ordinary" sequences.
2. Non-modifying functions that work on lists will also work on iterators.
A Better Solution

Iterators Are Sequences

def is_palindrome(a):
    return a == "".join(reversed(a))
A Better Solution

Iterators Are Sequences

```python
def is_palindrome(a):
    return a == ''.join(reversed(a))
```

sequence methods also work on iterators

```python
>>> iter = reversed("abracadabra")
>>> ".join(iter)
"arbadacarba"
```
Anagrams

An anagram is a type of word play, the result of rearranging the letters of a word or phrase to produce other words, using all the original letters exactly once – again, Wikipedia.
The Problem

Anagrams

An anagram is a type of word play, the result of rearranging the letters of a word or phrase to produce other words, using all the original letters exactly once – again, Wikipedia

Anagram Checking

How?
A Short Solution

def anagrams(a, b):
    return sorted(a) == sorted(b)
Using Strings as Sequences, Again

A Short Solution

def anagrams(a, b):
    return sorted(a) == sorted(b)

sorted also works on Strings

>> sorted("hello")
['e', 'h', 'l', 'l', 'o']
Part V

Modules
Module Primer

Structuring Code

To structure code and keep the size of single source code files at a reasonable level, Python programs are split up into modules.
Module Primer

Introduction to Python

Using Modules

Writing Modules

Creating a New Module

Exports & Names

Packages

The Module Search Path

Structuring Code

To structure code and keep the size of single source code files at a reasonable level, Python programs are split up into modules.

Modules are Files

A module always denotes one single file in the file system. However, a module can re-export any number of symbols from other modules.
Using Modules

Importing

To be able to use functions (or classes) from a module, it needs to be imported. Example: the built-in module `math`:

```python
import math
```

print math.sqrt(2)
Using Modules

Importing

To be able to use functions (or classes) from a module, it needs to be imported. Example: the built-in module `math`:

```python
import math
```

Dotted Names

From now on, any object in this module can be called using dotted notation:

```python
print math.sqrt(2)
```
Namespaces

No Name Clashes

Different modules can have members with the same name. In a program, both members can be accessed using the fully-qualified (i.e. dotted names) and no conflicts will appear.
Namespaces

No Name Clashes

Different modules can have members with the same name. In a program, both members can be accessed using the fully-qualified (i.e. dotted names) and no conflicts will appear.

Importing Names From a Modules

Sometimes, all or some symbols from a module should be injected into the current namespace:

```python
from math import *
from random import shuffle
```

Now, all functions from `math` and the functions `shuffle` can be accessed without a specifying a prefix.
Module Creation

Modules Aren’t Special

Every file with valid Python source code and a name that ends with " .py " can also be used as a module.
Module Creation

Modules Aren’t Special

Every file with valid Python source code and a name that ends with "\.py" can also be used as a module.

Module Locations

- same directory as the executed program
- module search path
Module Creation

Modules Aren’t Special

Every file with valid Python source code and a name that ends with ".py" can also be used as a module.

Module Locations

- same directory as the executed program
- module search path

What Happens on Import?

When a module is imported, it is loaded and executed by the Python interpreter (almost) like a regular script.
Exported Symbols

Voluntary Export Restraints

Sometimes, it might not be wanted to export all symbols from a module in order to hide implementation details, prevent cluttering namespaces etc.
Exported Symbols

**Voluntary Export Restraints**

Sometimes, it might not be wanted to export all symbols from a module in order to hide implementation details, prevent cluttering namespaces etc.

**... Through Magic!**

The "magic" variable `__all__` may contain a list with names to be exported. Still, all other symbols in the module are accessible when the module is imported by its name.
**Naming Conventions**

**Some Privacy**

Specifiers starting with an underscore will be excluded when importing *. By convention, they are considered to be private members of the module, but still be accessible.
Some Privacy

Specifiers starting with an underscore will be excluded when importing *_. By convention, they are considered to be private members of the module, but still be accessible.

A Word of Warning

- Slogan: ”We’re all consenting adults here”
- No way to restrict access to module-private function
If a program or library gets too big to fit into one module, it is usually split up into packages, which is the logical grouping of several modules.
Packages

Packaging Modules

If a program or library gets too big to fit into one module, it is usually split up into packages, which is the logical grouping of several modules.

Creating a Package

- a directory with a name that is also a Python identifier
- in this directory, a file named `__init__.py`
Importing Nested Modules

To import a module from a package, you can use both import variants:

```python
import os.path
from os import path
```
The Module Search Path

By default, the Python interpreter will search the following locations:

- the current directory
- all directories in the environment variable PYTHONPATH
- the installation directory of the Python distribution