Python
An Introduction

Jörg Faschingbauer

www.faschingbauer.co.at

jf@faschingbauer.co.at

Table of Contents

FASCHINGBAUER

Overview

The Python Programming Language

FASCHINGBAUER

- *Interpreted*
  - No compiler (and entire toolchain) needed
  - Interpreter generates intermediate *byte code*
- *Object Oriented*
  - Classes/encapsulation, exception handling, ...
  - But not mandatory as in Java, for example
- *Interactive*
  - *Python prompt* — Interpreter's *interactive mode*
- *For beginners*
  - Simple syntax: *indentation* instead of explicit block markers
  - Consistent
  - "There's only one way to do it!"
- *Powerful*
  - Advanced language features: Iteration, `yield`, ...
  - Huge library — "Comes with batteries included"

A Little Bit of History

FASCHINGBAUER

- Written and conceived by Guido Van Rossum during the late eighties
  - Named after *Monty Python*
- First public release 1991 — version 0.9.0
  - Modern language attributes: classes, exceptions, modules, ...
- Version 1.5 (1997)
  - Major version for a longer time
  - Several useful features: keyword arguments, functional programming tools, name mangling/data hiding, ...
- Version 2.7 (2010)
  - Still backwards compatible with all previous versions
  - Last version of the 2.x series
  - Only fixes
  - Promised to be supported until 2020
- Version 3.0 (2008)
  - Incompatible in subtle ways

Guido Van Rossum

FASCHINGBAUER

- *Benevolent Dictator for Life* (BDFL)
- Oversees Python's development process
- Born 31 January, 1956 in the Netherlands
- Degree in Math and Computer Science (University of Amsterdam)
- Jobs permit at least 50% work on Python
    - Google
    - Dropbox

Overview

Hello World: Interactive Mode

FASCHINGBAUER

Interactive Mode

- Python interpreter, invoked without arguments
- "Shell prompt", just with Python
- Exit → type Strg + d (End-of-file)

```
$ python
Python 2.7.9 (default, Aug 15 2015, 22:03:50)
[GCC 4.8.4] on linux2
Type "help", "copyright", "credits" or "license" for ...
>>> print "Hello World"
Hello World
>>>
```

Hello World: Python 3

Major annoyance: Python 3 is not compatible with Python 2

- Breaking compatibility is not an easy decision
- Necessary (so they say) to clean up >20 years of dirt
- First hurdle: `print` is a *statement* in 2, and a *function* in 3

```
$ python3
Python 3.4.1 (default, Aug 15 2015, 22:12:12)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for ...
>>> print("Hello World")
Hello World
>>>
```

Hello World: Python 2 vs. Python 3

FASCHINGBAUER

Compatibility strategy: the `__future__` module

- Enable future features in current versions
- Clearly remains valid in the future version
- One of many strategies
- The remainder of the course will try to be ...
    - version agnostic
    - forward compatible
    - backward compatible

Tataa: the feature `print_function`!

```
$ python2
...
>>> from __future__ import print_function
>>> print("Hello World")
Hello World
>>>
```

Hello World: Script Files

The first and simplest program ...

hello-world.py

```
#!/usr/bin/python

# omitted from now on
from __future__ import print_function

print("Hello World")
```

Make it executable, execute ...

```
$ chmod +x hello-world.py
$ ./hello-world.py
```

Overview

FASCHINGBAUER

Syntax: Indentation (1)

FASCHINGBAUER

Blocks and indentation

- Statements that end with a ':' introduce a *block*
- Blocks are *indented*
- End of a block is end of indentation
- *No explicit block delimiters* (like '{', 'BEGIN', ...)
- Indentation is not only *Coding Style*, but also *Syntax*

Careful, you experienced programmers!

- New bug type: *Indentation Bug*

Syntax: Indentation (2)

FASCHINGBAUER

```
i = 0
while i < 42:
    print('Still not an answer: '+str(i))
    i = i+1
print('The answer is: '+str(i))
```

Keep in mind ...

- Indentation must be consistent *within one block*
- ... can be mixed otherwise
- Tune your editor's knobs accordingly!

Syntax: Statements and Lines

FASCHINGBAUER

Newline ends a statement ...

```
answer = 42
```

Except ...

Multiline statements

```
answer = str(42) + \
    ', but only most of the time'
```

Braces

```
print(
    "Hello",
    "World")
```

Brackets

```
message = [
    "Hello",
    "World"]
```

Fun

```
message = (
    "Hello " +
    "World")
```

Commandline Arguments

Python is lean:

- Very few *built-in* functionality (compared to other languages)
- Extension through *modules*
- First (and most used): `sys`

File `args.py`

```
#!/usr/bin/python
import sys
print(sys.argv[0])
print(sys.argv[1])
print(sys.argv[2])
```

```
$ ./x.py one argument
./x.py
one
argument
```

Comments vs. Documentation

FASCHINGBAUER

As in many other script languages ...

```
# this is a very important comment, which is
# definitely worth a read
```

Docstrings (slightly off-topic)

- First string in a function, module, class, or method
- Tools to generate documentation from it

```
def do_something(some_number):
    """ Doing something with a number """
    # some code here ...
```

```
>>> print(do_something.__doc__)
Doing something with a number
```

Overview

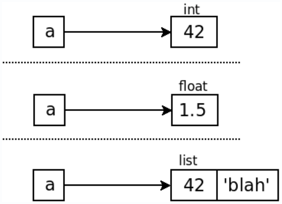
FASCHINGBAUER

Variables (1)

A variable is a name for ... something

- *Something* has a type
- ... but its name hasn't

```
>>> a = 42
>>> type(a)
<class 'int'>
>>> a = 1.5
>>> type(a)
<class 'float'>
>>> a = [42,'blah']
>>> type(a)
<class 'list'>
```



Variables (2)

FASCHINGBAUER

Python is a “dynamic language” (whatever that means)

- Names have no type
- Created when first assigned
- → *Runtime error* when accessed but not yet there
- … as opposed to *compiled languages* (whatever that means)

Naming rules: just like most other languages

- Start with Letters (Unicode since Python 3, ASCII in Python 2) or underscore
- Numbers in the following characters
- Case sensitive

Assignment Fun

FASCHINGBAUER

Multiple assignments in one statement

```
a, b, c = 1, "Eins", 1.0
a, b = b, a  # "swap"
```

- *Tuple unpacking*
- Important concept throughout the entire language
- $\rightarrow$ later

Assignment has a value

```
a = b = c = 1
```

- Assignment is *right associative*
- $\implies$ `a, b, c` are assigned '1'

Assignment Details

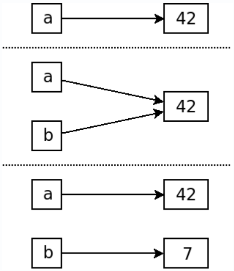
More than one ever wants to know ...

- Day-to-day programming does not need to know
- Good to know when something goes wrong
- Only valid for *immutable* types (`int`, `float`, `str`)

```
a = 42
b = a
b = 7
```

Or equivalently ...

```
a = 42
b = 42
b = 7
```



Overview

Overview

FASCHINGBAUER

Numbers

FASCHINGBAUER

Numbers are simplest …

- Integer (**int**) — *sign* is irrelevant
- Floating point (**float**)
- Complex (**complex**)
- Boolean (**bool**)

More powerful types …

- *Sequences* with very powerful operations
    - Immutable sequences: Strings, Bytes, Tuples
    - Lists
- *Sets*
- *Mappings*: *key* to *value*

Integer Numbers

FASCHINGBAUER

Range ...

- Represent numbers in an *unlimited* range — limited by available memory only

Integer literals ...

- Decimal: `1234`, `-1234`
- Octal: `01234` == `1*8**3+2*8**2+3*8**1+4*8**0` == 668
- Hexadecimal: `0x1234` == `1*16**3+2*16**2+3*16**1+4*16**0` == 4660
- Binary: `0b100110`

Integer Numbers: Comparison

FASCHINGBAUER

Comparison operators

<	less than
<=	less or equal
>	greater than
>=	greater or equal
==	equal
!=	not equal

Integer Numbers: Arithmetic

FASCHINGBAUER

Arithmetic operators

+	addition
-	subtraction
*	multiplication
/	division
//	floor division
%	modulo
**	exponentiation
-	negation (unary)

Shortcut: self modification (not only for the + operator)

```
i = i + 7
i += 7
```

Operator Precedence

Boring but important: precedence rules

- Exponentiation comes first (*binds strongest*)
- Negation
- *, /, % (left associative)
- +, - (binary operators)
- Comparison operators

Not boring — necessary in programming

- If in doubt, use explicit braces: `2 * 7 % 3 != 2 * (7 % 3)`
- If not in doubt, think about colleagues
    - If in doubt, use explicit braces

Floating Point Numbers

FASCHINGBAUER

Floating point vs. Integer

- Operators listed above also valid for floating point numbers
- Not unbounded
    - ... otherwise π would consume all memory

Literals

- Decimal point: `3.14159265359`
- Exponent: `2.3e12`, `1.5e-34`

Numbers: Python2 vs. Python 3 (1)

Incompatibility alert!

- There is no *pure integer division in 3*
- `int` only if possible
- `float` otherwise
- … *as opposed to 2*

Reason:

- Python is also a beginners language
- There are many other incompatibilities as well
- … the entire object model has changed

Python 2

```
>>> 3/2
1
>>> type(3/2)
<type 'int'>
```

Python 3

```
>>> 3/2
1.5
>>> type(3/2)
<class 'float'>
```

Numbers: Python2 vs. Python 3 (2)

FASCHINGBAUER

General advice regarding numbers

- Do not rely on the division operator (/) to do *floor division*
    - Portably, `3/2 != 1`
    - Not easy when coming from Java or C
    - ... or just about any other language
- Don't differentiate between `int` and `float`
- Use *explicit floor division*, *//*
    - Portably, `3//2 == 1`

Overview

FASCHINGBAUER

Strings: Python 2

Python 2 strings ...

- A string could have just about any encoding
- Strings were raw bytes, basically
- Everybody had to know *where* the string came from
- Could be ASCII, could be Unicode, could be bytes, could be ...
- Type `unicode` — added as an afterthought
- File I/O done *without* an idea of encoding

Problems ...

- Implicit conversions back and forth
- Clearly defined but not at all obvious
- → Mixing text and binary

Strings: Python 2 — Confusion

```
>>> type('abc')
<type 'str'>
>>> 'abc'
'abc'
>>> len('abc')
3
```

- That was easy
- ASCII

```
>>> type('äöü')
<type 'str'>
>>> 'äöü'
'\xc3\xa4\xc3\xb6\xc3\xbc'
>>> len('äöü')
6
>>> 'äöü'[0]
'\xc3'
```

- Content comes from terminal
- → UTF-8 (in my case)
- Umlauts are 2 bytes in UTF-8
- → Gosh!

Strings: Python 2 — unicode (1)

FASCHINGBAUER

Good news

```
>>> type(u'äöü')
<type 'unicode'>
>>> u'äöü'
u'\xe4\xf6\xfc'
>>> len(u'äöü')
3
>>> u'äöü'[0]
u'\xe4'
```

- Explicit type `unicode`
- Content is typed
- (I still don't get it)

Strings: Python 2 — `unicode` (2)

FASCHINGBAUER

Bad news

```
>>> type(u'abc' + 'def')
<type 'unicode'>
>>> type(u'abc' + b'def')
<type 'unicode'>
```

- Can be mixed with `str`
- Can be mixed with `bytes` (which is another afterthought)
  - $\rightarrow$ Semantics not entirely clear
- $\rightarrow$ *Chaos*
- $\rightarrow$ *Bugs, bugs, bugs ...*

Strings: Python 3

FASCHINGBAUER

Strings are always Unicode — Basta!

- Major reason for the 2 to 3 move
    - Python 2 Unicode is a mess
- No `unicode` type anymore
- *No mixing* of `str` and `bytes`
- *Sources* which create strings know about encodings — and create Unicode strings accordingly
    - File I/O

Python 3, Generally

FASCHINGBAUER

Which version should I choose

- Answer 1: Python 3
- Answer 2: unless you have a compelling reason not to
    - Large Python 2 codebase
    - Ancient distro version (though there are Python 3 packages available for most)

So much for Python 2 vs. 3 ...

Datatype Conversions

FASCHINGBAUER

Conversion between types ...

```
>>> str(42)
'42'
>>> int('42')
42
>>> int('10', 16)
16
>>> float('12.3')
12.3
>>> int(12.3)
12
```

- Conversions
- Better viewed as *constructors* of the corresponding types
- Common theme across the language

Overview

FASCHINGBAUER

Complex Datatypes By Example: List, Tuple

FASCHINGBAUER

Typical "sequence" types ...

List

```
l = list()
l = [1,2,3]
l.append(4)
l.extend([5,6,7])
l += [8,9]
new_l = l + [10,11]
```

- *Mutable*: can be modified *in-place*
- Type: `list`

Tuple

```
t = tuple()
t = (1,2,3)
t = (1,)
new_t = t + (4,5)
```

- *Immutable*: cannot be modified, only copied
- Type: `tuple`

Complex Datatypes By Example: Dictionary

FASCHINGBAUER

Dictionary

```
>>> d = dict()
>>> d = {1:'one', 2:'two'}
>>> d[2]
'two'
>>> d[3] = 'three'
>>> 3 in d
True
>>> del d[3]
>>> 3 in d
False
```

- *Associative array*
- Key → value mapping
- Common operations: insert, remove, query

Complex Datatypes By Example: Set

FASCHINGBAUER

Set

```
>>> s = set()
>>> s = {1,2,3}
>>> 1 in s
True
>>> s.add(4)
>>> s
{1, 2, 3, 4}
>>> s.remove(1)
>>> 1 in s
False
```

- Bag of elements
- No value like dictionary
- Membership test is the most important operation

Overview

FASCHINGBAUER

Boolean Values

FASCHINGBAUER

Boolean: the last of the simple ones

```
>>> 1 < 2
True
>>> 'X' == 'U'
False
```

- Values `True` and `False`
- Result of comparison operators
- Used with control flow statements (`if`, `while`)
- $\rightarrow$ later

Boolean Operators

Usual operators ...

- `L and R`: `True` if both `L` and `R` evaluate to `True`
- `L or R`: `True` if `L` or `R` evaluate to `True`
- `not X`: `True` if `X` evaluates to `False`

Short circuit evaluation: operands are only evaluated until the expression's value is clear

- `L and R`: if `L` is `False`, then the expression cannot become `True` anymore → `R` not evaluated
- `L or R`: if `L` is `True`, ...
- → important when `L`, `R` are functions with *side effects*

Overview

FASCHINGBAUER

The if Statement

FASCHINGBAUER

Conditional code execution ...

if

```
if i <= 3:
    print(i)
```

else (optional)

```
if i <= 3:
    print(i)
else:
    print('many')
```

elif (optional)

```
if i == 1:
    print('1')
elif i == 2:
    print('2')
elif i == 3:
    print('3')
else:
    print('many')
```

Overview

FASCHINGBAUER

Exercises

FASCHINGBAUER

1. In the interactive interpreter, create an empty list. Append to it values of types
   - Integer
   - Floatingpoint
   - Boolean
   - List
   - Tuple
   - Set
   - Dictionary

   Does it work? If yes, print the list using the `print()` *function*.
2. Do the same in an executable Python program
3. What happens when you access a non-existent dictionary member?
4. Write a program that takes a single digit as commandline parameter. Print the English word for that digit.

Overview

Looping Constructs

FASCHINGBAUER

Program flow is rarely linear ...

- Branches → `if/elif/else`
- Repeated execution → *loops*
- Python has only two looping constructs
- `while`
  - Handcrafted *loop condition*
  - → very "verbose" coding
  - Most general looping construct
- `for`
  - *iteration* over something sequencish
  - Iteration ... generators ... *yield* ... outright genius!
  - → later

while Loops

FASCHINGBAUER

General form of a while loop

```
while condition:
    statements
```

- *condition* is a boolean expression
- *statements* is an indented block of … well … statements
- Block is executed while *condition* holds

Example: sum of numbers 1..100

```
sum = 0
i = 1
while i <= 100:
    sum += i
    i += 1
```

break and continue

Fine grained loop control ...

- break ends the loop
- continue ends the current loop and continues with the next — evaluating the condition

```
while True:
    line = sys.stdin.readline()
    for c in line: print(c, ord(c))
    if len(line) == 0:
        # eof seen
        break
    if line.strip() == '':
        # ignore empty lines
        continue

    # ... do something ...
```

Esoteric Feature: while/else

FASCHINGBAUER

Loops can have an else clause

- Entered when loop terminates “naturally”
- … *not* terminated by a break
- Natural while loop termination: loop condition evaluates to False

```
i = 0
while i < 100:
    i += 1
    number = random.randrange(0,1001)
    if number == 42:
        break
else:
    print('no answer found')
```

Overview

FASCHINGBAUER

Exercises

1. Write a program that takes an integer commandline parameter and checks whether that number is prime!

Overview

FASCHINGBAUER

Sequential Datatypes

FASCHINGBAUER

Sequential Datatypes are a “sequence” of elements

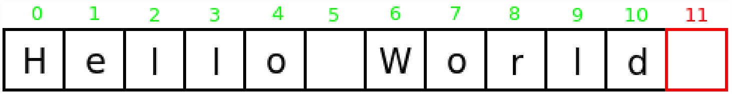
- Strings: sequence of Unicode “code points”
- Lists: *mutable* sequence of elements of *any* type ($\rightarrow$ recursive)
- Tuples: like lists, but *immutable*
- Binary data ...
  - Bytes: like strings, only binary — there is no *encoding*. *Immutable*
  - Byte arrays: *mutable* arrays of raw bytes
- Common set of operations
  - Indexing
  - Concatenation
  - Several specialities: slicing ...
- Very powerful (albeit a bit hard to read)

Sequence Elements

FASCHINGBAUER

Elements are numbered

- Starting at *zero*



Sequence Membership

FASCHINGBAUER

The in operator

```
>>> 2 in ['one', 2, 'three']
True
>>> 3 in ['one', 2, 'three']
False
>>> 'three' in ['one', 2, 'three']
True
>>> 'three' not in ['one', 2, 'three']
False
```

- Cool for short sequences
- *Sequential search*
- $\rightarrow$ probably not the right datastructure for searches

Sequence Multiplication

FASCHINGBAUER

String multiplication

```
>>> 'blah' * 5
'blahblahblahblahblah'
```

Arbitrary sequence multiplication

```
>>> [1, 2, 3] * 5
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> ['one', 2, 'three'] * 3
['one', 2, 'three', 'one', 2, 'three', 'one', 2, 'three']
```

Overview

Indexing (1)

Accessing the n-th element is straightforward ...

```
>>> text = "Hello World"
>>> text[0]
'H'
>>> text[6]
'W'
>>> text[-1]
'd'
>>> text[-4]
'o'
>>> text[len(text)-1] == text[-1]  # AAH!!
True
```

Indexing (2)

Same with other sequences ...

```
>>> a_list = ['Peter', 'Paul', 'Mary']
>>> a_list[0]
'Peter'
>>> a_list[-1]
'Mary'
```

```
>>> a_tuple = (1, 'one', 1.0)
>>> a_tuple[0]
1
>>> a_tuple[-1]
1.0
```

Slicing: Cutting Out

Extracting part of a sequence

```
>>> text = "Hello World"
>>> text[0:5]
'Hello'
>>> text[:5]
'Hello'
>>> text[6:11]
'World'
>>> text[6:]
'World'
>>> text[6:-1]
'Worl'
>>> text[-5:-1]
'Worl'
```

Slicing: Step Width

FASCHINGBAUER

Killer feature: slices with step width

```
>>> text = "Hello World"
>>> text[0:7:2]
'HloW'
>>> text[::2]
'HloWrd'
>>> text[:-6:2]
'Hlo'
>>> text[::-1]
'dlroW olleH'
```

Slice Assignment

FASCHINGBAUER

Sub-slice assignment

```
>>> l = [2, 3, 'a', 'b', 7]
>>> l[2:4] = [4, 5, 6]
>>> l
[2, 3, 4, 5, 6, 7]
```

Prepending

```
>>> l[:0] = [0, 1]
>>> l
[0, 1, 2, 3, 4, 5, 6, 7]
```

Appending (but see `list` methods `append()` and `extend()`)

```
>>> l[len(l):] = [8, 9]
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Overview

FASCHINGBAUER

Iteration over ... Something

FASCHINGBAUER

Iteration: a central concept everywhere

- Programs build and manipulate data
- ... and occasionally (most often?) iterate over data
- → Specialized looping construct: `for`

```
for name in ['Caro', 'Johanna', 'Eva', 'Jörg']:
    print(name)
```

- `name`: *loop variable*
- Valid only within the loop body
- Bound to the current element in the list, four times in a row
- A list is *iterable* — many other types participate in this game

break, continue, else

Just as with while: usual looping features

- break ends the loop — else clause not executed
- continue executes block with next element (if any)

```
haystack = ['straw', 'mouse', 'straw', 'needle', 'straw']
for item in haystack:
    if item == 'needle':
        break
else:
    print("couldn't find needle")
```

Overview

FASCHINGBAUER

Iteration over Numbers: `range`

FASCHINGBAUER

Rare: iteration using indexed access

Indexed access in C

```
char hello[] = "Hello World";
for (int i=0; i<sizeof(hello)-1; i++)
    printf("%d\n", hello[i]);
```

- Rarely needed in Python
- Iteration over *data*
- If needed: sequence of integer numbers

```
hello = 'Hello World'
for i in range(len(hello)):
    print(ord(hello[i]))
```

range: Definition

FASCHINGBAUER

The `range` function produces numbers ...

- `range(100)` produces `0, 1, 2, ... 99`
- `range(5, 100)` produces `5, 6, 7, ... 99`
- `range(5, 100, 2)` produces `5, 7, 9, ... 99`

Produces?

- Result cannot easily be a list: `range(10**9)`

```
>>> type(range(10**9))
<class 'range'>
```

- *Generates* numbers on demand
- $\rightarrow$ *“Generator”*

range: Python 2 vs. Python 3

FASCHINGBAUER

Incompatibility alert:

- Python 2: `range(10**9)` *would* explode!
- Heritage of the old Pre-Generator days
- → Python 2's `xrange()` is a generator

If one wants a list in Python 3 (unlikely) ...

```
l = list(range(10**9))
```

Overview

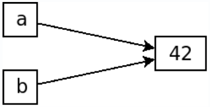
FASCHINGBAUER

Immutability: Numbers

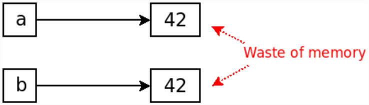
Numbers are immutable ...

- Object of type `int` with value 42
- Variable a points to it ("gives it a name")
- The object cannot change its value — there is no method to modify an integer object
- → The latter situation is equivalent to the former (which is the implementation)

```
a = 42
b = a
```



```
a = 42
b = 42
```



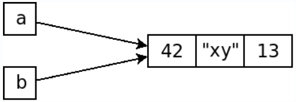
Immutability: Tuples

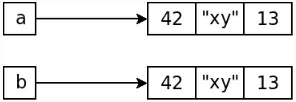
FASCHINGBAUER

Same with tuples

- Like lists, but *immutable*
- No way to modify a tuple
  - No appending
  - No slice assignment
  - No nothing
- So both of these are equivalent
  - To the user, `b` *is a copy of* `a`

```
>>> a = (42, "xy", 13)
>>> b = a
```



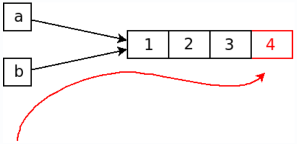


Mutability: Lists (1)

Lists are mutable ...

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> b.append(4)
>>> b
[1, 2, 3, 4]
>>> a
[1, 2, 3, 4]
```

- Objects can be modified
- E.g. by using append()



Mutability: Lists (2)

FASCHINGBAUER

Danger ...

- Take care when passing complex data structures
- Not passed *by copy* (as in C++)
- Passed *by reference* (as in Java)
- Make a copy if needed

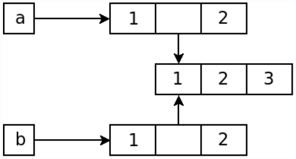
Copying a list

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> a.append(4)
>>> b
[1, 2, 3]
```

Shallow Copy

```
>>> a = [1, [1, 2, 3], 2]
>>> b = a[:]
>>> b
[1, [1, 2, 3], 2]
>>> a[1].append(4)
>>> a
[1, [1, 2, 3, 4], 2]
>>> b
[1, [1, 2, 3, 4], 2]
```

```
>>> a[1] is b[1]
True
```



- Only first level copied
- “Shallow copy”
- `a[1]` is a *reference*
- `is`: *object identity*

Deep Copy

FASCHINGBAUER

Solution: not easy

- Recursive structure traversal
- Handling every possible type
- Dedicated module in the standard library: copy

```
>>> import copy
>>> a = [1, [1, 2, 3], 2]
>>> b = copy.deepcopy(a)
>>> a[1] is b[1]
False
```

Overview

FASCHINGBAUER

Why Functions?

What is a function?

- Sequence of statements
- Parameterizabe
- Can have a return value
- $\rightarrow$ Can be used as an expression

Why would one want to do this?

- Code structuring
- Readability
- Maintainability
- Code reuse
- $\rightarrow$ Libraries

An Example

```
def maximum(a, b):
    if a < b:
        return b
    else:
        return a

max = maximum(42, 666)
```

- `def`: introduces function definition
- `maximum`: function name
- `a` and `b`: parameters
- `return`: ends the function — the *value* when used as expression

Sidenote: Pure Beauty

FASCHINGBAUER

There is nothing special about functions

- def is a *statement*
- Evaluated during regular program flow, just like other statements
- Creates a function object
- Points a *variable* to it — the function's name

```
>>> type(maximum)
<class 'function'>
>>> a = maximum
>>> a(1,2)
2
```

Parameters and Types

There is no compile-time type check

- For good or bad
- `maximum(a,b)`: can pass anything
- … provided that `a` and `b` can be compared using $<$
- "Late binding" $\rightarrow$ runtime error
- $\rightarrow$ More testing required
- $\rightarrow$ Unit testing, module `unittest`

```
>>> maximum(1, '1')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in maximum
TypeError: unorderable types: int() < str()
```

Default Parameters

For the most common case, *default values* may be specified ...

```
def program_exit(message, exitstatus=0):
    print(message, file=sys.stderr)
    sys.exit(exitstatus)

program_exit('done')
```

Default parameters must come at the end of the parameter list ...

Syntax error

```
def program_exit(exitstatus=0, message):
    ...
```

Default Parameters: Pitfalls

Attention: mutable default parameters may not do what one expects ...

```
def f(i, x=[]):
    x.append(i)
    return x

print(f(1))
print(f(2))
```

Produces ...

```
[1]
[1, 2]
```

Reason: default value for a parameter *is part of the function object* → retains its value across calls

Keyword Arguments

FASCHINGBAUER

Long parameter lists ...

- Easy to confuse parameters
- Unreadable
- Unmaintainable

Function call with keyword arguments

```
def velocity(length_m, time_s):
    return length_m / time_s

v = velocity(2, 12) # what?
v = velocity(time_s=2, length_m=12)
```

- $\rightarrow$ Very obvious to the reader!

Local and Global Variables

Best explained using examples ...

x only visible/alive inside f()

```
def f():
    x = 100
    return x
```

Error: no x found anywhere

```
def f():
    return x
```

Using x from global scope

```
x = 100
def f():
    return x
```

x defined *globally* when f() called first time

```
def f():
    global x
    x = 100
    ...
```

Overview

FASCHINGBAUER

Exercises

1. Modify the prime number detection program from one of the previous exercises: make the prime number detection a function, and call the function instead. The function (`is_prime()` is a likely name) takes a number, and returns a boolean value as appropriate.
2. Write a function `uniq()` that takes a sequence as input. It returns a list with duplicate elements removed, and where the contained elements appear in the same order that is present in the input sequence. The input sequence remains unmodified.
3. Write a function `join()` that takes a string list `strings` and a string `separator` as parameter. It joins `strings` together into a single string, using `separator` as a separator. For example,
   - `join(['Hello', 'World'], '-')` returns `'Hello-World'`
   - `join(['Hello'], '-')` returns `'Hello'`

Overview

FASCHINGBAUER

String Delimiters

Delimiters: double quotes ("...") or single quotes ('...'), as needed

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Escape Sequences

FASCHINGBAUER

Newline, embedded in string

```
>>> print('first line\nsecond line')
first line
second line
```

More (but not all) escape sequences

`\n`	Linefeed, ASCII 10
`\r`	Carriage return, ASCII 13
`\t`	Tab
`\b`	Backspace
`\0`	ASCII 0 in octal
`\130`	ASCII 88 ('X') in octal
`\x58`	ASCII 88 ('X') in hexadecimal

Raw Strings

FASCHINGBAUER

Unwanted escaping (Doze pathnames)

```
>>> print('C:\some\name')
C:\some
ame
>>> print(r'C:\some\name')
C:\some\name
```

Unwanted escaping (regular expressions)

```
regex = re.compile(r'^(.*)\.(\d+)$')
```

Multiline Strings

FASCHINGBAUER

Escaping newlines is no fun ...

```
print("""\
Bummer!
You messed it up!
""")
```

will produce ...

```
Bummer!
You messed it up!
```

- Note how the initial newline is escaped → “line continuation”
- Newline must immediately follow backslash

More String Tricks

FASCHINGBAUER

String literal concatenation

```
>>> 'Hello' ' ' 'World'
'Hello World'
```

String literal concatenation (multiple lines)

```
>>> ('Hello'
... ' '
... 'World')
'Hello World'
```

Overview

C-Style Formatting (1)

FASCHINGBAUER

Good old C: `%[flags][width][.precision]type`

Program

```
int i = 42;
float f = 3.14159265359;
printf("%07d, %8.4f\n", i, f);
```

Output

```
0000042,   3.1416
```

Same in Python, using the % operator

```
>>> '%07d' % 42
'0000042'
>>> '%07d, %8.4f' % (42, 3.14159265359)
'0000042,   3.1416'
```

C-Style Formatting: Conversions

Frequently used conversions

`s`	String
`c`	Single character
`d`	Integer (decimal)
`o`	Integer (octal)
`x`	Integer (hexadecimal lowercase)
`X`	Integer (hexadecimal uppercase)
`f`	Floating point, exponential format (lowercase)
`F`	Floating point, exponential format (uppercase)
`%`	The % sign itself

C-Style Formatting: Flags

FASCHINGBAUER

Frequently used flags

#	Octal or hex integer conversions: `0x...` prefixes
0	Pad with '0' characters
–	Left alignment
+	Print sign even if positive
(space)	Print space in place of sign if positive

C-Style Formatting: Examples

FASCHINGBAUER

```
>>> '%#5X' % 47
' 0X2F'
>>> '%5X' % 47
'   2F'
>>> '%#5.4X' % 47
'0X002F'
>>> '%#5o' % 25
' 0o31'
>>> '%+d' % 42
'+42'
```

```
>>> '% d' % 42
' 42'
>>> '%+2d' % 42
'+42'
>>> '% 4d' % 42
'  42'
>>> '% 4d' % -42
' -42'
>>> '%04d' % 42
'0042'
```

The format Method

Problems with C-style formatting

- Not flexible enough (as always)
- Positional parameters only
- Parameter position must match occurence in format string

A better (?) way of formatting

```
>>> '0 {0:05d}, 1 {1:8.2f}, 0 again {0}'.format(42, 1.5)
'0 00042, 1     1.50, 0 again 42'
>>> 'a {a:05d}, b {b:8.2f}, a again {a}'.format(a=42, b=1.5)
'a 00042, b     1.50, a again 42'
```

- More → RTFM

Overview

FASCHINGBAUER

Batteries Included: Checks

FASCHINGBAUER

Lots of small checks (returning boolean) — for example ...

- `'...'.isspace()`: contains only whitespace
- Character types
  - `'...'.isalpha()`
  - `'...'.isalnum()`
  - `'...'.isdigit()`
- Case tests
  - `'...'.isupper()`
  - `'...'.islower()`
- `'...'.isidentifier()`: a valid python identifier (e.g. variable name)
- Lots of others → save work and RTFM prior to coding

Batteries Included: Search

FASCHINGBAUER

Substring search ...

- `'...'.count(s)`: number of occurences of s
- `'...'.startswith(s), .endswith(s)`
- `'...'.find(sub[, start[, end]])`: find sub, starting at start (default 0), ending at end (default `len()`)
  - end is *exclusive* → `'...'[start:end]`
  - Returns index, or -1 if not found
- `'...'.index(sub[, start[, end]])`: like find, but raises exception if not found
- `'...'.rfind(sub[, start[, end]])`: from the end
- `'...'.rindex(sub[, start[, end]])`: from the end

Substring Search: Examples

```
>>> '/etc/passwd'.startswith('/etc/')
True
>>> 'notes.txt'.endswith('.txt')
True
>>> 'this is a thistle with many thorns'.count('th')
4
>>> 'haystack containing needle and straw'.find('needle')
20
>>> 'haystack containing needle and straw'.find('mouse')
-1
>>> 'haystack containing needle and straw'.index('mouse')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

Split and Join (1)

FASCHINGBAUER

- *Very* common operations
- Error prone → writing them is a major annoyance
- Off-by-one errors

split() and join()

```
>>> 'one:two:three'.split(':')
['one', 'two', 'three']
>>> ':'.join(['one', 'two', 'three'])
'one:two:three'
```

Not off-by-one

```
>>> ':'.join([])
''
>>> ':'.join(['one'])
'one'
```

Split and Join (2)

Split at most 2 fields

```
>>> 'one:two:three:four'.split(':', 2)
['one', 'two', 'three:four']
>>> 'one:two:three:four'.rsplit(':', 2)
['one:two', 'three', 'four']
```

Real life example: /etc/passwd

```
>>> username,rest = 'jfasch:x:1000:...'.split(':', 1)
>>> username
'jfasch'
>>> rest
'x:1000:1000::/home/jfasch:/bin/bash'
```

Overview

FASCHINGBAUER

Character Encodings

FASCHINGBAUER

Problem ...

- Files (and networks, and ...) contain arbitrary bytes
- Files don't have an idea of their content
- $\rightarrow$ Content can be anything
  - Raw bytes
  - Plain 7-bit ASCII
  - ISO 8859-1
  - One of 2156 Chinese (multibyte) character sets
  - One of 1375 Japanese (multibyte) character sets
  - UTF-8, UTF-16, UTF-32
  - Many *many* more ...

Solution ...

- Unicode — one encoding to rule them all
- Internally, Python *strings* are sequences of Unicode *code points*

Strings and Encodings

FASCHINGBAUER

Where does the data come from and go to?

- Programmer has to know what the source contains, and act accordingly
- Raw bytes $\rightarrow$ create `bytes` objects
- Strings $\rightarrow$ which encoding?
  - Email: MIME headers ($\rightarrow$ `email` module)
  - Files: specify `encoding` parameter at `file` object creation ($\rightarrow$ later)
  - Otherwise: read byte data and convert to string objects

At the programmer's responsibility!

- Has always been programmer's responsibility
- Python 3 just doesn't let you mix `str` and `bytes`

From Raw Bytes to Strings (1)

Pre-Unicode: ISO/IEC 8859-1 ("Latin-1") for Mid-European alphabet

Jörg, as read from a file with unknown encoding

```
>>> joerg_raw = b'J\xf6rg'
>>> type(joerg_raw)
<class 'bytes'>
```

- File happens to be Latin-1 encoded
- `\xf6` is "ö" in Latin-1
- ... but that information isn't there → binary

From Raw Bytes to Strings (2)

Transformation to string should be done as early as possible

- Everything's clear if one *knows* what's in
- $\rightarrow$ Transformation to Unicode (rules them all)
- $\rightarrow$ Nobody *has* to know anymore what's in

Transfer raw bytes into string

```
>>> joerg = str(joerg_raw, encoding='iso-8859-1')
>>> type(joerg)
<class 'str'>
>>> joerg
'Jörg'
```

From Strings to Raw Bytes

FASCHINGBAUER

Internal string representation is Unicode

- No-one cares (has to care)
- Unicode is a set of numbers, not a concrete encoding

“ö” is obviously multibyte in UTF-8

```
>>> joerg.encode('utf-8')
b'J\xc3\xb6rg'
```

“ö” is unknown in China

```
>>> joerg.encode('big5')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'big5' codec can't encode ....
```

Source File Encoding

Question: how are string literals encoded?

- Default: ASCII
- $\rightarrow$ umlauts not properly encoded in strings
- Unless otherwise specified

Explicit source encoding

```
#!/usr/bin/python3
# -*- encoding: utf-8 -*-

print('Jörg')
```

Overview

Exercises

1. Write a program that receives any number of arguments and prints them out right justified at column 20.

Overview

List Access

FASCHINGBAUER

In addition to sequence access …

- `L.append(elem)`: append `elem` to the list
- `L.extend(l)`: extend `L` with another sequence `l`
- `L.insert(i, elem)`: insert `elem` at position `i` (same as `L[i:i] = elem`)
- `L.pop(i)`: remove element at `i` from the list (and return its value)
- `L.sort()`: sort the list *in place*. Elements must be comparable → careful with mixed lists!
- `L.reverse()`: reverses the list *in place*
- `sorted(L)`: return a sorted copy of the list
- `reversed(L)`: returns a reversed copy of the list

List Methods: Examples

```
>>> l = [3, 2, 5]
>>> l.append(3)
>>> l
[3, 2, 5, 3]
>>> l.extend([3, 2])
>>> l.sort()
>>> l
[2, 2, 3, 3, 3, 5]
>>> l.reverse()
>>> l
[5, 3, 3, 3, 2, 2]
>>> sorted(l)
[2, 2, 3, 3, 3, 5]
```

List Comprehension

FASCHINGBAUER

The best way to write good code is to write as little code as possible ...

Best explained by example

```
>>> [i**2 for i in [1, 2, 3]]
[1, 4, 9]
```

Traditional alternative

```
def square_numbers(numbers):
    ret = []
    for i in numbers:
        ret.append(i**2)
    return ret
sqn = square_numbers([1,2,3])
```

Overview

FASCHINGBAUER

Dictionaries

Associative arrays ...

- Stores pairs of *key* and *value*
- Keys are unique
  - → no two keys with the same value can exist in the same dictionary object
- Fast lookup
- Internally realized as a *hash table*
  - Keys are not sorted
  - No deterministic iteration possible

Dictionary Access

FASCHINGBAUER

`d[key] = value`	Insert (or overwrite) `value` under `key`
`d[key]`	returns value of `key` (or raises exception)
`d.get(key)`	returns value of `key` (or `None` if not there)
`d.get(key,defval)`	returns value of `key` (or `defval` if not ther
`del d[key]`	remove entry for `key` (exception if not ther
`d.keys()`	iterable over keys
`d.values()`	iterable over values
`d.items()`	iterable over data as `(key,value)` tuples
`len(d)`	number of entries (as with all non-scalar ty
`d.setdefault(key,defval)`	return value if there, else insert `defval` and
`d.update(other)`	merge dictionary `other` into this
`key in d`	does `key` exist in `d`?
`key not in d`	does `key` not exist in `d`?

Examples: Simple Access

FASCHINGBAUER

Literal, insertion, access

```
>>> d = {} # empty
>>> d = {'one': 1, 'two': 2}
>>> d['one']
1
```

Nothing there

```
>>> d.get('one')
1
>>> d.get('three')
None
>>> d['three']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'three'
```

Examples: Shortcuts

Shortcuts for what would otherwise be too much code ...

Default without modification

```
>>> d.get('three', 3)
3
>>> d.get('three')
None
```

Default with modification

```
>>> d.setdefault('three', 3)
3
>>> d['three']
3
```

Dictionary Iteration (1)

- Iteration is a fundamental concept in Python
- ... even more so in Python 3
- → *compatibility alert!*

Python 3

```
>>> d.keys()
dict_keys(['three', 'one', 'two'])
>>> list(d.keys())
['three', 'one', 'two']
```

Python 2

```
>>> d.keys()
['three', 'two', 'one']
>>> d.iterkeys()
<dictionary-keyiterator object at 0x7ff2e8753418>
```

Dictionary Iteration (2)

FASCHINGBAUER

Iteration over values

```
>>> list(d.values())
[3, 1, 2]
>>> list(d.items())
[('three', 3), ('one', 1), ('two', 2)]
```

- Wait: `d.item()` lets me iterate over tuples ...
- Why shouldn't I use *tuple unpacking* then?

The entire power of Python

```
for key, value in d.items():
    ...
```

Building Dictionaries

FASCHINGBAUER

```
>>> d = {}
>>> d = {1: 'one', 2: 'two'}
>>> d = dict()
>>> d = dict({1: 'one', 2: 'two'})
>>> d = dict([('one', 1), ('two', 2), ('three', 3)])
```

Overview

FASCHINGBAUER

Sets

- Unordered collection of distinct objects
- $\rightarrow$ *set* in a mathematical sense
- Membership tests
- Addition and removal of elements
- Mathematical operations, like ...
  - Intersection
  - Union
  - Difference

Operations on Sets (1)

Test operations

`x in s`	Is `x` member of `s`
`x not in s`	`in`, negated
`s1 == s2`	`True` if both contain the same elements
`s1 != s2`	...
`s.isdisjoint(other)`	Does `s` have no elements in common with `other`
`s1 <= s2`	Is `s1` a subset of `s2`?
`s1 < s2`	Is `s1` a *strict* subset of `s2`?
`s1 >= s2`	Is `s1` a superset of `s2`?
`s1 > s2`	Is `s1` a *strict* superset of `s2`?

Operations on Sets (2)

Building sets from other sets

`s1	s2`	Union
`s1 & s2`	Intersection
`s1 - s2`	Difference
`s1 ^ s2`	Symmetric difference

- All operations available as `|=` (for example)

Constructing sets

```
>>> s = {1, 2, 3}
>>> s = set([1, 2, 3]) # ... or any iterable
```

Overview

Python 2 vs. Python 3

Encoding, again: incompatibility alert!

- Python 2 already had types `str` and `bytes`
- ... it just didn't make a difference
- Files are inherently binary, at the lowest level
- ... and so were Python 2's files
- Python 3 won't let you mix `str` and `bytes`
- Hard rule: "Transform to string as early as possible"
- $\Longrightarrow$ Transformation must be done inside file I/O
- $\Longrightarrow$ Files know about their encoding
- $\Longrightarrow$ Python 2 vs. Python 3

Opening Files

FASCHINGBAUER

Files are opened to obtain a *handle*

```
f = open('/etc/passwd')
```

- `f` refers to an *open file*
- Buffered IO (as `stdio` in C)
- Read-only (the default)
- Python 3: UTF-8 encoded (the default, unless otherwise specified)
- $\rightarrow$ I/O is done in units of *strings*

Specifying an encoding

```
f = open('/etc/passwd', encoding='ascii')
```

Reading Files

FASCHINGBAUER

`f.read()`	reads entire file content
`f.read(n)`	reads `n` characters/bytes
`f.readline()`	reads a line (*including* the terminating linefeed)
`f.readlines()`	reads entire file $\rightarrow$ list of lines

Note the end-of-file condition

```
while True:
    line = f.readline()
    if len(line) == 0:
        break
    print(line)
```

Shorter but less resource-friendly

```
for line in f.readlines():
    print(line)
```

Reading Files: Pythonic

FASCHINGBAUER

Iteration is a central theme in Python

- Readability
- "Iterable": anything that can be iterated
- Many things can be iterated
- Fine-tunable behaviour and performance
- Why shoudn't we iterate files?

```
for line in f:
    print(line)
```

Writing Files (1)

FASCHINGBAUER

Open file *write-only*

```
f = open('/tmp/some-file', 'w')
```

Writing arbitrary content

```
f.write('arbitrary content')
```

Writing multiple “lines”

```
f.writelines(['one\n', 'two\n'])
```

Using `print()`, linefeed added automatically

```
print('one line (with automatic linefeed)', file=f)
```

Writing Files (2)

FASCHINGBAUER

The beauty of iteration (again) ...

- `writelines()` does *not* add linefeed (probably a misnomer)
- Items can come from any *iterable*
- $\rightarrow$ *Very* cool!

Copying a file the Pythonic way

```
src = open('/etc/passwd', 'r')
dst = open('/tmp/passwd', 'w')

dst.writelines(src)
```

File Modes

FASCHINGBAUER

Available `mode` characters

`r`	open for reading (default)
`w`	open for writing, truncating the file first
`x`	open for exclusive creation, failing if the file already exists
`a`	open for writing, appending to the end of the file if it exists
`b`	binary mode (no encoding and decoding)
`t`	text mode (default)
`+`	open a disk file for updating (reading and writing)

Combinations and their meanings

`w+`	read/write/truncate
`r+`	read/write (write pointer at beginning)
`a+`	read/write (write pointer at end)

Text vs. Binary Mode

FASCHINGBAUER

Python 3 is Unicode clean — for file I/O this means ...

- Cannot pass `bytes` to a file opened in text mode
- Cannot pass `str` to a file opened in binary mode
- Unless otherwise specified (`mode='b'`), files are in *text mode*

Python 2 is not Unicode clean

- `mode='b'` means "No stupid CR/LF conversion on Doze"
- `bytes` or `str`, noone cares

Standard Streams

FASCHINGBAUER

Good Ol' Unix ...

Number	POSIX Macro	Python equivalent
0	`STDIN_FILENO`	`sys.stdin`
1	`STDOUT_FILENO`	`sys.stdout`
2	`STDERR_FILENO`	`sys.stderr`

- Interaktive Shell: all three associated with terminal
- Standard input and output used for I/O redirection and pipes
- Standard error receives errors, warnings, and debug output

$\Longrightarrow$ Windows-Programmers: no errors, warnings, and debug output to *standard output*!!

Error and debug output goes to *standard error*

```
print('An error occurred', file=sys.stderr)
```

Overview

FASCHINGBAUER

Exercises (1)

FASCHINGBAUER

1. Write a program `wc.py` that takes a filename from the commandline and counts
   - Lines
   - Words
   - Characters

   and then outputs the gathered statistics to `stdout`
2. Write a program `revert.py` that takes a filename from the commandline, and outputs every line of the file with the line's characters reversed. (Take care to strip off the linefeeds, or otherwise the linefeed will come first in the reversed line.)
3. Write a program `distill.py` that takes a filename from the commandline, and outputs only those lines that are not empty or don't entirely consist of a Python style comment.

Exercises (2)

FASCHINGBAUER

1. Write a program `user.py` that takes one or more usernames from the commandline, looks them up in `/etc/passwd`, and prints out the user records one after the other. The program should be optimized for speed and read `/etc/passwd` only once. The user records are pre-parsed as follows: the metadata (UID, home directory, etc.) go in a dictionary

   ```
   { 'uid': 1000,
     'gid': 1000,
     'home': '/home/jfasch',
     'shell': '/bin/bash'
   }
   ```

   The user records are sorted into another dictionary, with the user's login name as the key. It is that dictionary where the lookup is performed.

Overview

Overview

FASCHINGBAUER

What's a Function?

First: what's a variable?

- A name that refers to something (here: an integer object)
- Created at first assignment

```
i = 1
```

Functions are no different ...

- The function's name refers to a *function object*
- ... it's just that object creation is done differently

```
def square(number):
    """
    return square
    of the argument
    """
    return number**2
```

Function Objects?

FASCHINGBAUER

square is a name that happens to refer to a function object ...

Object and its attributes

```
>>> square
<function square at 0x7fca2c785b70>
>>> square.__doc__
'\n    return square\n\tof the argument\n\t'
```

The “()” Operator

```
>>> square(3)
9
```

Function Objects! (1)

FASCHINGBAUER

Dynamic languages require care

```
>>> square = 1
>>> square(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

Assign one variable to another

```
op = square
op(3)
```

Function Objects! (2)

FASCHINGBAUER

Function as function argument

```
def forall(op, list):
    result = []
    for elem in list:
        result.append(op(elem))
    return result

print(forall(square, [1, 2, 3]))
print(forall(len, ["Joerg", "Faschingbauer"]))
```

This will output ...

```
[1, 4, 9]
[5, 13]
```

Batteries included: Python built-in function `map`

Overview

FASCHINGBAUER

Iteration in Python

- for loops are very common in Python
- They operate on *iterators*
- Just about any composite data type is *iterable*
  - Lists
  - Dictionaries
  - Strings
  - Files
  - ...

What's an Iterator?

FASCHINGBAUER

An iterator is an object that yields a *data stream* ...

- The `next()` method yields the next element in the stream
- If there is no next element, it raises the `StopIteration` exception

Question: where do iterators come from?
Answer: they are made by *iterables*

What's an Iterable?

Iterables are objects that support *iteration* (Gosh!)
Iterables that are built into Python are for example ...

- Sequence, tuple
- Dictionary (iteration yields key/value pairs)
- Set
- String
- File
- ... and many more ...

The Iterator Protocol (1)

FASCHINGBAUER

Technically speaking ...

- An *iterable* can make an *iterator* through the `__iter__()` method
- Not usually done by hand
- Done for me by `for` loop

```
for elem in iterable:
    ... do something with elem ...
```

The interpreter ...

- Creates an *iterator* before entering the loop (→ `__iter__()`)
- Calls `next()` on that iterator before every iteration
- Terminates the loop when `StopIteration` is caught

The Iterator Protocol (2)

FASCHINGBAUER

Manually

```
iterator = iter(iterable)
try:
    i = next(iterator)
except StopIteration:
    ...
```

- Often the calculation of the next element is complicated
- $\rightarrow$ object state has to be kept manually
- Coding iterables is no fun
- ... at least not without proper language support

Generators: Motivation

Examples of complicated iteration ...

- Traverse a binary tree in depth-first or breadth-first order
- Infinite sets like Fibonacci numbers

Stupid solution:

- Store result in a list
- Return the list
- $\rightarrow$ Problem with large iterables (Fibonacci?)
- $\rightarrow$ Best to *generate* on-demand

Generators: How?

A sample generator

```
def odd_numbers():
    i = 0
    while True:
        if i%2 != 0:
            yield i
        i += 1

for j in odd_numbers():
    print(j)
```

Observations

FASCHINGBAUER

- `odd_numbers` is *iterable*
- `yield` is magic
- Every function that calls yield is a *generator*
- Each call to `next(iterator)` (speak: execution of the `for` body) continues the function where `yield` left it.
- *This is outright genius!*

More on Generators

Python 2 to 3 transition

- `range()` is a generator in 3
- Python 2: returns a (temporary) list
- ... had to use `xrange()` to generate
- Many more places converted to generators

Standard library helpers

- `itertools`
- `operator`

Overview

Exercise: Fibonacci

Write a function that *generates* an infinite sequence of Fibonacci numbers!
Make the start values configurable!

Overview

FASCHINGBAUER

Object Oriented Programming

OO Principles

- *Procedural*: there's data, and there's code
- $\rightarrow$ relationship is not aways clear
- *OO*: data and code aggregated together, into *classes*
- $\rightarrow$ *Methods* operate on *objects* that have *members*
- *Encapsulation*: implementation is hidden from the public

End effect: you talk about your code in the same way that you program it

OO Everywhere

FASCHINGBAUER

Strings

```
s = 'Jörg'
enc_s = s.encode(encoding='utf-8')
```

Lists

```
list = ['Hello', 'World']
list.extend(['!'])
```

Batteries

```
from http.client import HTTPConnection
connection = HTTPConnection('www.google.com')
connection.connect()
```

The `class` Statement

FASCHINGBAUER

Defining a class: the `class` statement

```
class MakesNoSense:
    ...
```

- `class` creates a "class" object ($\rightarrow$ *Metaprogramming*)
- `MakesNoSense` is the name of a variable (that refers to the class object)
- $\rightarrow$ like with functions, the class object can be assigned, passed as parameter, ...

The Constructor

FASCHINGBAUER

```
class MakesNoSense:
    def __init__(self, parameter1, parameter2):
        ...

mns = MakesNoSense('Hello', 666)
```

- `__init__`: special method name → *constructor*
- *self*: the object being initialized/constructed
- Python does not require the name `self`, but it is "good style". IDE's may rely upon it, but no requirement otherwise.

Attributes/Members

```
class MakesNoSense:
    def __init__(self, parameter1, parameter2):
        self.member1 = parameter1
        self.member2 = parameter2


...
print(mns.member1)
mns.member2 = 42
```

- There is no *information hiding* in Python
- Members are visible to outside users
- ... by default at least

Attributes/Members: hiding

FASCHINGBAUER

```
class MakesNoSense:
    def __init__(self, parameter1, parameter2):
        self.__member1 = parameter1
        self.__member2 = parameter2

# error!
mns.__member2 = 42
```

- Python recognizes '__' as something special
- *Mangles* the name → visible as-is only within class's methods

Methods

FASCHINGBAUER

A **Method** is a function that "is called on an object" ...

```
class MakesNoSense:
    def __init__(self, parameter1, parameter2):
        self.__member1 = parameter1
        self.__member2 = parameter2

    def do_make_sense(self, value):
        print('I try to but fail: {} {} {}'.format(
            self.__member1, self.__member2, value))

mns = MakesNoSense(1, 'one')
mns.do_make_sense('bummer')
```

Overview

FASCHINGBAUER

Why Exceptions?

FASCHINGBAUER

Deal:

- Return <0 on error
- Caller has to check
- Caller has to pass error on

```
def do_much(this, that):
    if do_this(this) < 0:
        return -1
    if do_that(that) < 0:
        return -1
    return 0
```

```
def do_this(this):
    if this == 2:
        return -1
    else:
        return 9

def do_that(that):
    if that == 5:
        return -1
    else:
        return 'blah'
```

Exception Handling

FASCHINGBAUER

Plan is: write less code $\Longrightarrow$ cleaner code

```
def do_much(this, that):
    do_this(this)
    do_that(that)
```

```
try:
    do_much(1, 5)
except MyError as e:
    print('Error:', e.msg,
          file=sys.stderr)
```

```
def do_this(this):
    if this == 2:
        raise MyError('this is
    else:
        return 9

def do_that(that):
    if that == 5:
        raise MyError('that is
    else:
        return 'blah'
```

Exceptions

Exceptions are objects ...

- Python 2: can be anything
- Python 3: must be *derived* from class `BaseException`
    - User defined exception *should* be derived from `Exception`
- → *Object oriented programming*

```
class MyError(Exception):
    def __init__(self, msg):
        self.msg = msg
```

Catching All Exceptions

```
a_dict = {}
try:
    print(a_dict['novalidkey'])
except:   # KeyError
    print("d'oh!")
```

- Catches *everything* no matter what
- Hides severe programming errors
- → use only if you really know you want

```
try:
    print(nonexisting_name)
except:   # NameError
    print("d'oh!")
```

Catching Exceptions By Type

```
a_dict = {}
try:
    print(a_dict['novalidkey'])
except KeyError:
    print("d'oh!")
```

- `NameError` (and *most* others) passes through
    - ... and terminate the program unless caught higher in the call chain
- Very specific → best used punctually

Catching Exceptions By Multiple Types

FASCHINGBAUER

```
a_dict = {}
try:
    print(a_dict[int('aaa')])
except (KeyError, ValueError):
    print("d'oh!")
```

- (Btw, the exception list is an *iterable* of *type objects*)
- As always: reflect your intentions
- Is the handling the same in both cases?
    - I'd say very rarely

Storing the Exception's Value

FASCHINGBAUER

- Many exceptions' only information is their type
- $\rightarrow$ "A KeyError happened!"
- Sometimes exceptions carry additional information

```
class MyError(Exception):
    def __init__(self, msg):
        self.msg = msg

def do_something():
    raise MyError('it failed')

try:
    do_something()
except MyError as e:
    print(e.msg)
```

Order of Except-Clauses (1)

FASCHINGBAUER

- Except-Clauses are processed top-down
- $\rightarrow$ Very important when exceptions are related/inherited
- `MyError` *is a* `Exception`

```
class MyError(Exception):
    def __init__(self, msg):
        self.msg = msg

def do_something():
    raise MyError('it failed')
```

Order of Except-Clauses (2)

FASCHINGBAUER

Wrong

```
try:
    do_something()
except Exception as e:
    print('unexpected')
except MyError as e:
    print(e.msg)
```

- `MyError` is a `Exception`
- → eats all `MyError` objects
- → `MyError` never caught

Right

```
try:
    do_something()
except MyError as e:
    print(e.msg)
except Exception as e:
    print('unexpected')
```

Rule:

- Catch the *most specific* exception first

Overview

FASCHINGBAUER

Modules

FASCHINGBAUER

- Collection of ... well ... *objects* — e.g. classes, functions, variables
- Collected in a dedicated `.py` file
- Pulled in with the `import` statement

```
import sys
```

Searching `sys` ...

- In the directory where the *importer* lives
- Along the `PYTHONPATH` environment variable
- In the Python installation's module directories

Modules are Objects

- `import` makes a *module object* available under a name
- $\rightarrow$ a variable
- Contained names accessible through that variable
- $\rightarrow$ “Namespace”

```
import sys
...
sys.exit(42)
```

Other Forms (1)

FASCHINGBAUER

Pulling in a single symbol

```
from sys import exit
exit(42)
```

Massacre ...

```
from sys import *
exit(42)
```

- Pulls in *everything* into the importer's namespace
  - Well, except those names that start with an underscore
- Conflicts easily possible
- Importer's names are overwritten with conflicting names

Other Forms (2)

FASCHINGBAUER

Changing a module's name

```
import sys
my_sys = sys
del sys
```

Shorter ...

```
import sys as my_sys
```

Same with specific imports

```
from sys import exit as my_exit
my_exit(42)
```

Packages

FASCHINGBAUER

- *Package*: collection of modules (and further packages)
- “Subnamespace”

```
import os.path
path = os.path.normpath('a/../b')
```

```
from os.path import normpath
```

Executing Modules as Scripts

FASCHINGBAUER

- A module's name is its filename, with the `.py` extension stripped
- Available to the module in the variable `__name__`
- Can be used to decide if the module is being imported or executed as a script

Inside `mysupermodule.py`

```
def mysuperfunction(a, b):
    ...

if __name__ == '__main__':
    mysuperfunction(sys.argv[1], sys.argv[2]))
```

Package Structure

```
package/
+-- __init__.py
+-- subpackage1
|   +-- __init__.py
|   +-- module1.py
|   \- module2.py
\- subpackage2
    +-- __init__.py
    +-- module1.py
    \-- module2.py
```

- Top level directory `package/` found in module search path
- Each directory has file `__init__.py`
  - Disambiguation
  - Usually empty

Relative Imports (1)

```
package/
+-- subpackage1
    +-- module1.py
    \- module2.py
```

Problem: inside `module1.py`, I want to ...

- `import module2`
- *Not* search along the entire module search path
- I know that `module2` is next to me

```
from . import module2
```

Relative Imports (2)

```
package/
+-- subpackage1
    \-- module1.py
\- subpackage2
    \-- module1.py
```

Problem:

- `subpackage1/module1.py` wants to import `subpackage2/module1.py`
- ... and nothing else

```
from ..subpackage2 import module1
```

Overview

FASCHINGBAUER

Database Interfaces

FASCHINGBAUER

There are as many database interfaces for Python as there are databases ...

SQL:

- ODBC (generic)
- ADO (generic)
- MySQL
- Oracle
- PostgreSQL
- Informix
- SQLite
- ...

Others:

- BerkeleyDB
- ...

→ People want a common interface

DBAPI 2.0

FASCHINGBAUER

- Programming interface for SQL databases
- In fact only a *recommendation* for database interface authors
  - ... but there's the BDFL

Defines what a database interface has to have ...

- *Connection*: initial point of all database operations
- *Cursor*: context of a database operation. More than one cursor possible.
- *Data types*: e.g. `sqlite3.Date(1966,6,19)`

Caveat: Transaction Lifetime

FASCHINGBAUER

DBAPI module use the underlying database's "native interface" → *transaction semantics is not portable across different databases*

Neutral (DBAPI 2.0) Definition

- One connection has *at most one* transaction → transaction lifetime dictated by connection
- Once a cursor is created, a transaction is started
- The connection methods `commit()` and `rollback()` close a transaction
- A cursor's `.execute()` method creates a transaction if one does not exist
- Deleting a connection triggers a transaction's `rollback()` method
    - → *Don't forget* `connection.commit()`*!*

Caveat: Isolation

- Modifications on different cursors of the same connection are generally visible to each other
- Not all databases implement strong isolation among different *connections*
- *Isolation level* settings are specific to database implementations

Overview

FASCHINGBAUER

SQLite3

- Lightweight database implementation
- No big fat server, no client
- Relatively small C library — linkable by programs
- Used by ...
    - Android apps for configuration
    - Firefox to store history, bookmarks, whatever
    - ...
- Extremely cool for ...
    - Prototyping
    - Unit testing — *In-Memory database*
- *Bundled as DBAPI2 module in Python*

SQLite3 Connection

FASCHINGBAUER

Creating a database connection

```
import sqlite3
dbapi2 = sqlite3
connection = dbapi2.connect('/tmp/database')
```

Observations ...

- "Rename" module to `dbapi2` to ease porting to other DBAPI2 implementations (not necessary but cool)
- `dbapi2.connect('/tmp/database')` creates database if necessary $\rightarrow$ be careful
- `':memory:'` creates an in-memory database $\rightarrow$ cool for use in unit tests

SQLite3: Cursors and Statements

FASCHINGBAUER

Creating a cursor

```
cursor = connection.cursor()
```

Creating a table

```
cursor.execute('create table schwammerln ('
               '  name text, '
               '  typ text, '
               '  giftig boolean, '
               '  geniessbar boolean)')
connection.commit()
```

Observations ...

- It's SQL
- Commit is not necessary with SQLite3 — but could be with DBMS with a higher isolation level

SQLite3: Filling the Database

```
cursor.execute('insert into schwammerln '
 'values ("Steinpilz", "Roehren", 0, 1)')
cursor.execute('insert into schwammerln '
 'values ("Speisetaeubling", "Lamellen", 0, 1)')
cursor.execute('insert into schwammerln '
 'values ("Speitaeubling", "Lamellen", 0, 0)')
cursor.execute('insert into schwammerln '
 'values ("Eierschwammerl", "Lamellen", 0, 1)')
cursor.execute('insert into schwammerln '
 'values ("Teufelsroehrling", "Roehren", 1, 0)')
```

(connection.commit() as appropriate)

SQLite3: Queries — Result Set

FASCHINGBAUER

```
resultset = cursor.execute(
 'select * from schwammerln '
 'where typ = "Roehren"')
for row in resultset:
  print row
```

Output

```
(u'Steinpilz', u'Roehren', 0, 1)
(u'Teufelsroehrling', u'Roehren', 1, 0)
```

- A result set is *iterable*, and thus consumable *only once*

SQLite3: Bind Parameters

- Same statement, used repeatedly with varying *parameters*
- Naive implementation: Python string substitution
- Can be done better ...

```
cursor.execute('select * from schwammerln '
  'where typ = ?', ("Roehren",))
```

- Native interfaces are generally able to pre-calculate and optimize ("schedule") SQL statements
- SQL-Injection attacks

Overview

FASCHINGBAUER

DBAPI 2 Example: Postgres

FASCHINGBAUER

- Does not come with Python installation
- → http://initd.org/psycopg/
- Entry point: `connect()`
- Parameters best seen in the C-API documentation (http://www.postgresql.org/docs/8.3/static/libpq-connect.html)
- `connect(const char* conninfo)`: string containing `name=value` pairs
- → keyword arguments in `psycopg2`

```
import psycopg2
connection = psycopg2.connect(
    host='localhost',
    dbname='schwammerldb',
    user='ich',
    password='secret')
```

Overview

FASCHINGBAUER

SAX and DOM

FASCHINGBAUER

SAX

- Event-driven (elements start and end)
- Commonly used to parse long streams of structured data
- “De-facto” standard
- Available in multiple languages
- Python: `xml.sax`

DOM: “Document Object Model”

- Document available as a *tree*
- Programmatically navigable as a tree
- Relatively comfortable
- Python: `xml.dom`
- Problems
  - Only *relatively* comfortable
  - Not Pythonic enough

ElementTree

`xml.etree`: Python specific → *absolutely* comfortable

- Seamless integration in Python (→ iteration)
- A document is a tree, and trees are lists of lists
- XML attributes represented as dictionaries

→ simple!

A Very Simple Document

FASCHINGBAUER

Python code

```
from xml.etree.ElementTree import Element
element = Element("root")
child = Element("child")
element.append(child)
```

Or alternatively ...

```
element = Element("root")
SubElement(element, "child")
```

XML

```
<root>
  <child />
</root>
```

Attributes

- XML elements have attributes
- Python's XML elements have the attrib dictionary

```
element = Element("root")
child = SubElement(element, "child")
child.attrib['age'] = '15'
child = SubElement(element, "child")
child.attrib['age'] = '17'
```

```
<root>
  <child age="15" />
  <child age="17" />
</root>
```

Text (1)

FASCHINGBAUER

In XML documents, free text is permitted ...

- Inside one element
- After one element, but before the start of another element

Accordingly, Python elements have members ...

- `element.text`
- `element.tail`
- No text → `None`

Text (2)

FASCHINGBAUER

```
element = Element("root")
child = SubElement(element, "child")
child.text = 'Text'
child.tail = 'Tail'
```

```
<root><child>Text</child>Tail</root>
```

Careful with indentation

- Whitespace, linefeed etc. is text, no matter what
- `str.strip()` may be helpful

Writing XML Documents

FASCHINGBAUER

- We have created `Element` objects
- Added child elements
- Now how do we create XML?
- Wrap into `ElementTree` — a helper

```
from xml.etree.ElementTree import ElementTree
tree = ElementTree(element)
tree.write(sys.stdout) # oder file(..., 'w')
```

- Output is very tight
- Text is preserved as-is
- Pretty output would be incorrect
  - Linefeed and indentation is *text*

Reading XML Documents

FASCHINGBAUER

This is simple ...

```
from xml.etree.ElementTree import parse

tree = parse(sys.stdin)
for child in tree.getroot():
    age = child.attrib.get('age')
    if age is not None:
        print age
    if child.text is not None:
        print child.text
```

Overview

FASCHINGBAUER

Overview

Test Driven Development

FASCHINGBAUER

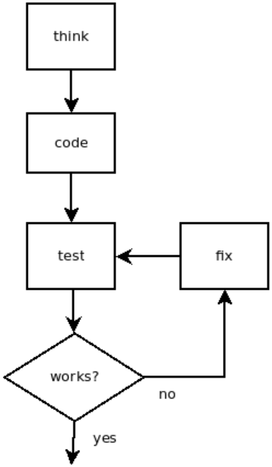
A simple idea … but first the problem …

- New code is written and tested since ages
    - Bugs are fixed until it works
    - Testing mainly done manually
    - Standalone test programs, or …
    - … mostly the entire target application
- Existing code breaks once it is modified (law of nature)
    - Breakage not easily detected
    - *Fear!*
    - $\Longrightarrow$ nobody ever modifies existing code
    - $\Longrightarrow$ software starts to rot once it has been written

Development — Traditional Approach

Traditional Approach

- Think about the design
- Come up with a decision
- Code it
- See if it works
- Fix
- (etc.)



Traditional Approach — Problems

FASCHINGBAUER

So what are the core problems?

- Before a modification ...
  - How do I know my solution will be ok?
  - How will it feel? Will it be usable?
  - Am I (and others) comfortable with it?
- After a modification ...
  - It is impossible to decide if everything still works
  - What is the definition of *everything*?
  - What is the definition of *works*?
  - What are the costs to decide that?
  - *What are the costs if we do only manual testing*?
  - *What is the state of the code? What about refactoring?*
- After the release ...
  - We curse at the testers that they do a bad job!

Test Driven Development — Principles (1)

FASCHINGBAUER

What if we were able to test everything automatically?

- Modifications could be done *without any fear*
  - *"Regression"*: new term for that kind of bug
  - Something that worked before a modification but doesn't afterwards
- Ongoing refactoring possible $\rightarrow$ no code smells
- New features would bring new tests
  - The *Everything* grows over time

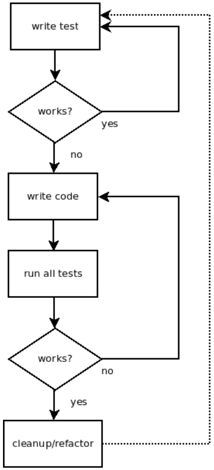
But: the Everything is now defined as ...

- Production code
- Test code

Test Driven Development — Principles (2)

Test Driven Development

- New "development process"
- Tests come first
- → "Requirements phase"
- Have you ever read a requirements document *after* coding was done?
- → Tests fail initially



Test Driven Development — Benefits? Caveats?

FASCHINGBAUER

What does it bring, what does it cost?

- More work initially — so much for sure
- Investment into the future
- More code can be done
- Not at all easy to convince people of it

Big caveat

- Tests belong to the code
- *No way* moving on without!
- $\Longrightarrow$ Have to take care of the tests

Overview

FASCHINGBAUER

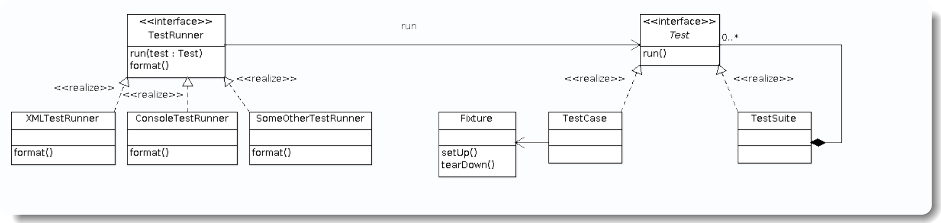
Origins

Unittest frameworks — where they come from

- `SUnit`, 1998. By Kent Beck in Smalltalk.
- `JUnit`, 2001. Ported from Smalltalk to Java, by Kent Beck and Erich Gamma.
  - Gained wide popularity by Kent Beck's book
- From then on ported to almost every language — commonly known as `xUnit`
  - Python: `PyUnit`, then became part of the Python library, module `unittest`
  - C++: `Boost.Test`, `CppUnit`, Google Test, ...
  - All the newer languages: Ruby, Rust, Go, ...
  - COBOL

xUnit Structure — Overview

FASCHINGBAUER



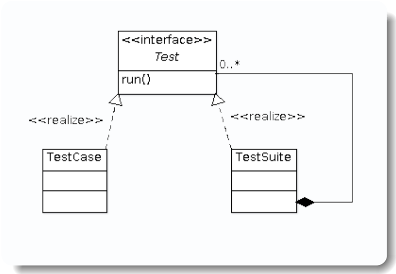
- TestCase: one test that is written. Here's the most code.
- TestSuite: composition of many test cases, for structural purposes.
- Fixture: defined environment of a TestCase
- TestRunner: runs a Test (Suite or Case), collects and presents results.

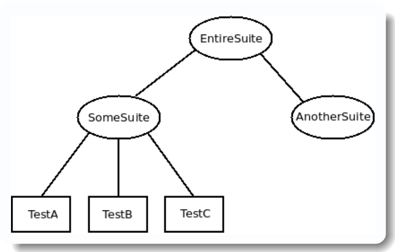
xUnit: TestCase and TestSuite

FASCHINGBAUER

Suites: recursive test structure

- Derive from `TestCase` to *implement* tests
- Use `TestSuite` objects to structure tests hierarchically
- Run a subset of all tests
- The *Composite Pattern* in use ...
- Not available in every xUnit incarnation

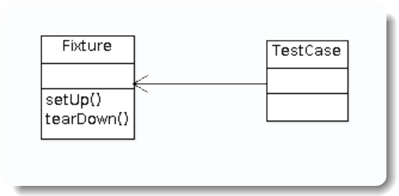




xUnit: TestCase and Fixture

Fixture: defined test environment

- Multiple tests start from the same state → common *Fixture*
- Method `setUp()` — establishes known state to start tests from. Examples: well-known/required database content, files have to be present, ...
- Method `tearDown()` — deallocates resources. For example: cleanup database, remove files, ...



Implementation:

- Python: class that contains test methods
- C/C++: weird macros to setup objects and associations

xUnit: TestCase and *Assertions*

FASCHINGBAUER

Test code checks for failure: Assertions

- Varying multitude of assertions to draw from
- Records test failure in some test result, for later reporting
- Abort the test case → failure
- Variation: *non-fatal* assertions

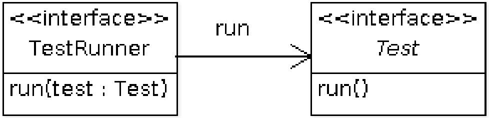
```
container.insert(100)
container.insert(200)
self.assertEqual(len(container), 2)
```

```
self.assertAlmostEqual(1/3, 0.333, 2)
```

xUnit: TestRunner

Running all tests: `TestRunner`

- `TestRunner` usually instantiated in main programs
- During running a test ...
  - Fixtures are prepared (`setup()`, `tearDown()`)
  - Results are collected
  - Failure or success
- After all tests have run ...
  - The result has to be presented
- (Sidenote: do you know the *Strategy Pattern*?)



Overview

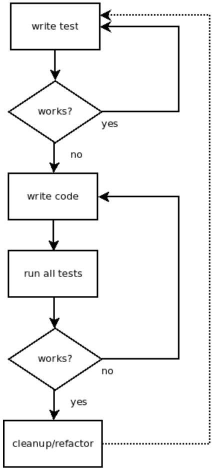
FASCHINGBAUER

The "Process"

FASCHINGBAUER

Test Driven Development is ... well ...

- Not a full process
- The basis of all "agile" processes
  - Anybody doing Scrum these days?
- It's *Software done right*
- It's about continuous investment and taking out



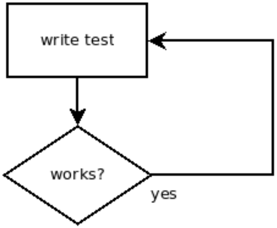
The “Requirements Phase”, New Code

Writing new code in a test driven way ...

- Nothing is clear from the beginning
- ... not even the problem

To get hold of the problem ...

- Write code that wouldn’t compile (there’s no solution yet)
- ... but gives you an impression of how a solution could look like
- Talk to people about proposed solution
- $\rightarrow$ “Finding the interface”
- *This is the first test*
- “Test First Development”



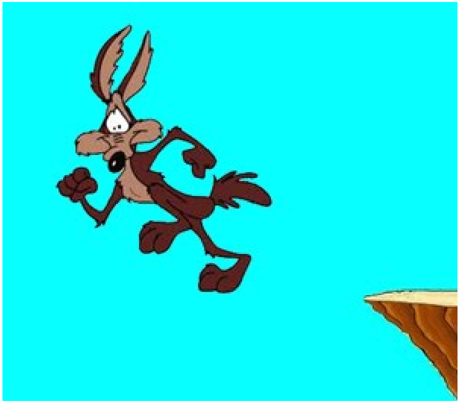
The “Requirements Phase”, Existing Code

Modifying existing code, to add features or change behavior ...

- Find the test suite for the module in question
  - $\rightarrow$ *structure* is important
- Add a new test for the new feature, making clear exactly what is wanted
  - The new test naturally fails, as always
- Modify code
- Run *all* tests
- Repeat

Caveats (1)

Take care of your tests! If your tests are suddenly gone, your code is alone ...



Caveats (2)

FASCHINGBAUER

- Tests are what ensure your code's value
- *You can do more valuable code with tests and TDD*
- Test code is no different from "real" code
  - $\rightarrow$ Subject to bitrot
- *"Lost Tests Syndrome"*: keep your hands off manual test suite arrangement
  - $\rightarrow$ Varying support from frameworks

Caveats (3)

But:

- Nobody tests the tests
    - *false impression*: “it’s only tests”
- *Structure* is important
- *Easy running* is important — everybody has to know how
- *Easy running*: avoid big dependencies — nobody will want to setup database infrastructure

Overview

Simplest Example

```
import unittest
class MyTestCase(unittest.TestCase):
    def runTest(self):
        self.assertEqual(1, 2)
c = MyTestCase()
unittest.TextTestRunner().run(c)
```

```
FAIL: runTest (__main__.MyTestCase)
----------------------------------------------------------
Traceback (most recent call last):
  File "/tmp/x.py", line 6, in runTest
    self.assertEqual(1, 2)
AssertionError: 1 != 2
```

Using a Fixture

Problems ...

- Cleanup after test failure
- Setup before test begin
- $\rightarrow$ formalize (prepare and release) a controlled environment for the test body

```
class MyTestCase(unittest.TestCase):
    def setUp(self):
        self.__db = create_database()
        fill_test_data(self.__db)
    def tearDown(self):
        remove_database(self.__db)
    def runTest(self):
        ...
c = MyTestCase()
unittest.TextTestRunner().run(c)
```

Multiple Test Cases With Same Fixture

FASCHINGBAUER

- A single `runTest()` method is not sufficient in most cases
    - A fixture's purpose is to serve multiple related test cases
- → test case with multiple test methods
- → *Test Suite*

```
class MyTestCase(unittest.TestCase):
    def setUp(self): ...
    def tearDown(self): ...
    def testFeature1(self): ...
    def testFeature2(self): ...
suite = unittest.TestSuite()
suite.addTest(MyTestCase('testFeature1'))
suite.addTest(MyTestCase('testFeature2'))
unittest.TextTestRunner().run(suite)
```

Auto Recognizing Test Methods

FASCHINGBAUER

Problems:

- Two steps: *write* test case and *add* test case
- → /me writes test, but forgets to add to suite
- → *Lost Test Syndrome*

```
class MyTestCase(unittest.TestCase):
    def setUp(self): ...
    def tearDown(self): ...
    def testFeature1(self): ...
    def testFeature2(self): ...
suite = unittest.TestLoader().\
    loadTestsFromTestCase(MyTestCase)
unittest.TextTestRunner().run(suite)
```

The Meat of a Test

Enough structure, now for the real test code ...

```
class MyTestCase(unittest.TestCase):
    def testSomething(self):
        self.failIf(1 == 2, "OMG!")
```

There's more:

- `failUnless(2 == 2)`
- `failUnlessEqual(2, 2)`
- `failIfEqual(2, 3)`
- `failUnlessAlmostEqual(2.12345, 2.123, 3)`
- `failUnlessRaises(IOError, file('/'))`

Recommendations

FASCHINGBAUER

A few recommendations, out of personal experience ...

- If tests become a burden, then you've messed it up!
- Tests should live *near* the code
  - ... but not *in* it
- Code must not use test code!
- Structure your tests (test suites) like your package structure
- *Test First Development* — adding tests afterwards is rarely fun
- There is no *Design for Testability* — sound design is always testable.
- It's easy to become an addict!

Overview

FASCHINGBAUER

Python Documentation

The best-documented language that I ever came across ...

- `python.org`: main python site
- `docs.python.org`
  - Browsable, searchable
  - Download tarball, unpack, bookmark to local
  - → easy offline operation (Javascript must be enabled though)

Notes