Introduction to Python

From the very basics to little advanced

Gabor Cseh – Lectures on Modern Scientific Computing (16. Nov. 2016)

ALWAYS LOOK ON THE BRIGHT SIDE OF LIFE…

- What is Python?
- Where does it come from?
- WHY?
- How?
- When?
- What does it do?

WHAT IS PYTHON?

- A programming language
- Named after Monty Python
- Multiple implementations (*C*, Java, Python,
  .NET, Common Lisp etc.)
- Multiparadigm: object-oriented, imperative, functional, procedural, reflective
- Created in 1989 by Guido van Rossum (BDFL)
- Open Source since 1991

Guido van Rossum,
creator of Python

- 1.0 Release in 1994
- 3.0 Release in 2008
- Latest stable release: 3.5.2 (6th Jul. 2016 – we use this!)

USING PYTHON: PROS AND CONS

- Easy to learn and read
- Good help system
- Interfacing (C/C++, Fortran etc.)
- Fast coding (prototyping)
- Concise (fewer lines of code, than C/C++, Java etc.)
- Extensive standard library
- Lots of quality packages for different usecases
- Lots of good tutorials and books

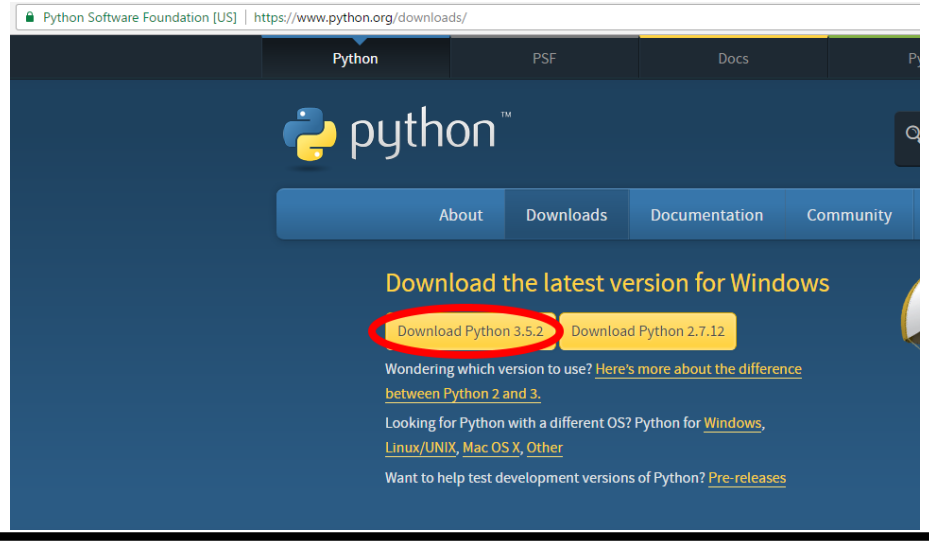
- Speed (interpreted → slow, but there are some workarounds)
- Lack of support for mobile computing and browsers
- Dynamically typed (more testing necessary)
- Multithreading restrictions (GIL) → workarounds exist

INSTALLATION

Linux

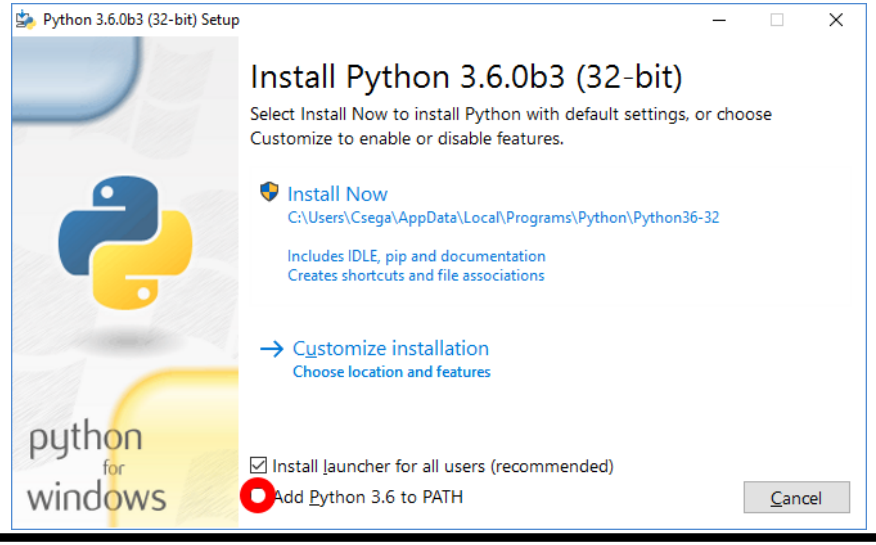
Probably you already have Python 3 installed (try > python3 in command prompt)

If not, use your package manager (sudo apt-get install python3 in Debian-like systems).



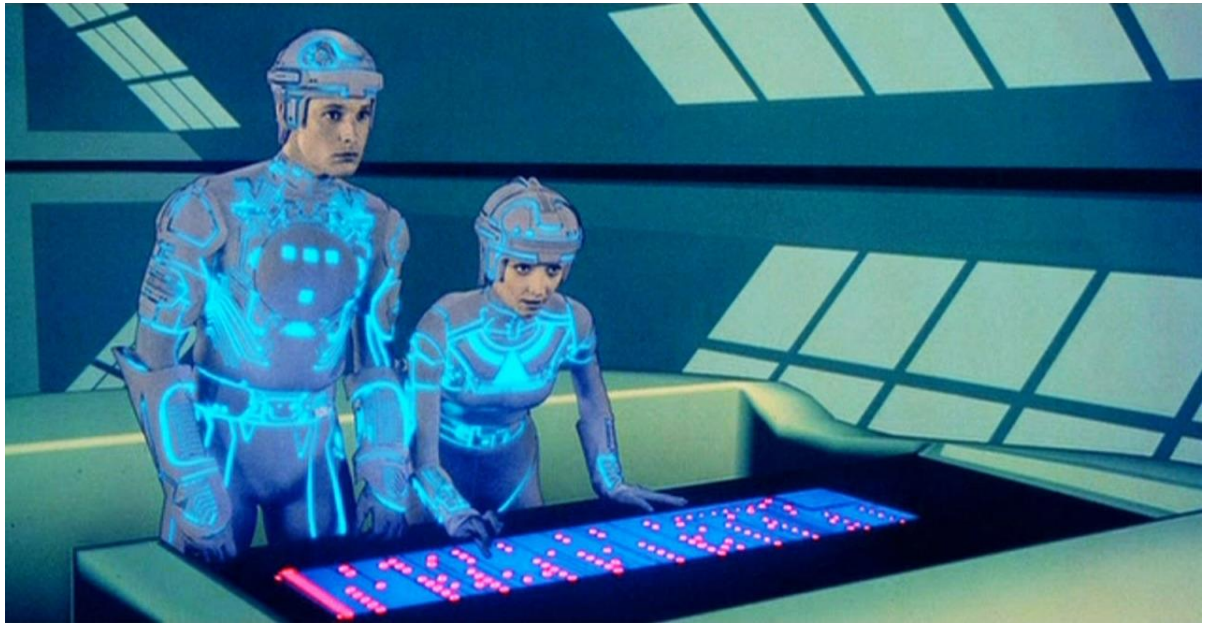
Windows

- Go to https://www.python.org/downloads/
- Choose „Download Python 3.5.2”
- Next-Next-Finish…
- Don’t forget to add it to the PATH!



VIRTUAL ENVIRONMENTS

- Python is coming with virtualenv, a tool to create isolated Python environments.
- Good to separate your projects, which need disjunct modules and/or require different versions of the modules.
- Windows (if Python is installed correctly and added to the PATH)
  - python –m venv myenv
  - ./myenv/Scripts/activate.bat
  - ./myenv/Scripts/deactivate.bat
- Linux
  - sudo apt-get install python3-venv
  - python3 –m venv myenv
  - source myenv/bin/activate
  - deactivate



PACKAGE MANAGEMENT

- Python has a lots of modules (libraries written by others than the Python Core Development Team).
- Only the standard library (which is very rich in features) comes preinstalled. The others (mainly numpy, scipy and matplotlib for scientific purposes) have to be installed separately.
- Linux
  - Use your package manager or pip. The typical packages names are python-[package name] or python3.x-[package name].
- Windows and Linux
  - pip install [package name] (you have to use pip3 under Linux)
  - Advantage of pip: in an isolated virtual environment, you can install packages separately with this tool.

THE ADVENTURE BEGINS…

- We won’t cover every aspects of Python.
- It is only a handful of selected aspects.
- Hopefully useful…

PYTHON AS A CALCULATOR

```
>>> 1 + 2 – 3 * 4 / 5
0.6
>>> 2**3  # (power – same as pow(2, 3))
8
>>> 4 / 5 (does not matter, that these are integers!)
0.8
>>> 4. // 5. (does not matter, that these are floats!)
0.0 (floor division)
>>> 4 % 3 (modulo)
1
```

```
>>> import math  # importing a module!
>>> math.pi
3.141592653589793
>>>  # sin, cos, tan, exp, log, log10, erf etc.
```

Wait! What?

Different libraries called modules.
They provide data types, classes, methods, constants etc.
We can import „factory” standard modules or our own code.

A LITTLE PHILOSOPHY…

>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

BASIC DATATYPES

```
# Int (arbitrary length)
>>> a = 2
# Float (actually, double called float)
>>> a = 2.1
# Complex
>>> re = 3
>>> im = 4
>>> a = complex(re, im)
>>> a.conjugate()
>>> (3-4j)
# Boolean (and, or, not, True, False)
>>> True and False
False
```

```
# String
>>> á = "árvíztűrőtükörfúrógép-ÁRVÍZTŰRŐTÜKÖRFÚRÓGÉP"
# Tuple
>>> a = (42, 137)
# List
>>> a = [1,2,3,4]
# Dictionary
>>> a = {'first': 1, 'second': 2}
```

YES! Everything is in UTF-8!

BASIC DATATYPES

```
# Int (arbitrary length)
>>> a = 2
# Float (actually, double called float)
>>> a = 2.1
# Complex
>>> re = 3
>>> im = 4
>>> a = complex(re, im)
>>> a.conjugate()
>>> (3-4j)
# Boolean (and, or, not, True, False)
>>> True and False
False
```

```
# String
>>> á = "árvíztűrőtükörfúrógép-ÁRVÍZTŰRŐTÜKÖRFÚRÓGÉP"
# Tuple
>>> a = (42, 137)
# List
>>> a = [1,2,3,4]
# Dictionary
>>> a = {'first': 1, 'second': 2}
```

YES! Everything is in UTF-8!

Set, Range, Byte objects (byte, bytearray, memoriview) etc.

BASIC DATATYPES - STRINGS

```
>>> a = „Introduction to Python”
>>> len(a)
22
>>> a.lower()
„introduction to python”
>>> a.upper()
„INTRODUCTION TO PYTHON”
```

```
>>> a.split(„ „)
[„Introduction”, „to”, „Python”]
>>>”-”.join(a.split(„ „))
„Introduction-to-Python”
>>> a.startswith(„Intro”)
True
>>> a.endswith(„Python”)
True
```

BASIC DATATYPES - STRINGS

```
>>> a.replace(„Python”, „strings”)
„Introduction to strings”

>>> a = „Introduction to”

>>> a += „Python”
„Introduction to Python”
```

```
>>> „to” in a
True
>>> a = „-”
>>> a*20
„--------------------”
```

PRINTING AND FORMATTING

```
>>> print(„Hello World!”)
„Hello World!”
>>> print(12)
12
>>> a = [1,2,3,4,5]
>>> print(a)
[1, 2, 3, 4, 5]
>>> print("The answer is: ", 42)
The answer is:      42
>>> print("The answer is: "+str(42))
The answer is: 42
```

PRINTING AND FORMATTING

```
>>> print('The value of PI is roughly %5.3f.' % 3.14159)
```

The value of PI is roughly 3.142.

```
>>> print("The value of PI again: {:.3}.".format(3.141592))
```

„The value of PI again: 3.14.

```
>>> print("Python is {} to {}.".format('fun', 'learn'))
```

Python is fun to learn.

```
>>> print("Let's see how {1} the rabbit {0} goes!".format('hole', 'deep'))
```

Let's see how deep the rabbit hole goes!

```
>>> print('To {where} and {where2}!'.format(where='infinity', where2='beyond'))
```

To infinity and beyond!

BASIC DATATYPES – LISTS, TUPLES

```
>>> a = [1, 2, 3, 4]
>>> a[3]
4
>>> a[4] = 5
Traceback (most recent call
last):
File "<pyshell#33>", line 1, in
<module>
    a[4] = 5
IndexError: list assignment
index out of range
```

```
>>> a.append(5)
>>> a.append(„string”)
>>> a.append([7, 8, 9])
>>> a
[1, 2, 3, 4, 5, 'string', [7, 8, 9]]
>>> one, two, _, *others, last = a
>>> one
1
>>> two
2
>>> others
[4, 5, 'string']
>>> last
[7, 8, 9]
```

This is called (un)packaging

Tuples are basically immutable lists (which makes them a bit faster). Packaging also works with them.

TRY-EXCEPT-FINALLY

How to handle exceptions, errors etc.?

```
>>> a = 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> try:
        a = 1/0
    except ZeroDivisionError as e:
        print(e)
    finally:
        a = ‚inf'
        print('Phew!')
```

```
>>> a = 'apple'
>>> if type(a) is str:
        raise AttributeError

Traceback (most recent call
last):
  File "<stdin>", line 2, in
<module>
AttributeError
```

OTHER LIST OPERATIONS

```
>>> a.insert(0, 1)
[1, 1, 2, 3, 4, 5, 'string', [7, 8, 9]]

>>> a.count(1)
2

>>> a.pop()
[7, 8, 9]

>>> a
[1, 1, 2, 3, 4, 5, ‚string’]

>>> a.remove(1)
[1, 2, 3, 4, 5, 'string']

>>> a.reverse()
['string', 5, 4, 3, 2, 1]
```

```
>>> a = [1,2,3,4,5, „string”]

>>> a.sort()
Traceback (most recent call last):
    File "<pyshell#86>", line 1, in <module>
    a.sort()
    TypeError: unorderable types: int() < str()

>>> a = [3,1,5,4,7,9]

>>> a.sort()
[1, 3, 4, 5, 7, 9]

>>> a = ['c', 'a', 'b', 'definition', 'zooming',
'yaml']

>>> a.sort()
['a', 'b', 'c', 'definition', 'yaml', 'zooming']
```

OTHER LIST OPERATIONS + SLICING

```
>>> a = [3, 1, 2, 6, „string”]
>>> a.index(6)
3
>>> a.insert(0, [1, 2, 3])
[[1, 2, 3], 3, 1, 2, 6, 'string']
>>> a.extend([4, 5, 6])
[[1, 2, 3], 3, 1, 2, 6, 'string', 4, 5,
6]
>>> a + [7, 8, 9]
[[1, 2, 3], 3, 1, 2, 6, 'string', 4, 5,
6, 7, 8, 9]
>>> a = [1, 2, 3]
>>> a * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
>>> a = list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[2:6]
[2, 3, 4, 5]
>>> a[:3]
[0, 1, 2]
>>> a[3:]
[3, 4, 5, 6, 7, 8, 9]
>>> a[-1]
9
>>> a[1:7:2]
[1, 3, 5]
>>> a[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

First index is included, second is not!

BONUS!

```
>>>
a[10:]
[]
```

DICTIONARIES

```
>>> a = dict()
>>> a[„one”] = 1
>>> a[„two”] = 2
{'two': 2, 'one': 1}
>>> a.get(‚one’)
1
>>> a.keys()
dict_keys(['two', 'one'])
>>> a.values()
dict_values([2, 1])
for key in a.keys():
    print(key, a[key])
```

SETS

No duplicates!

```
>>> a = set([1, 2, 3, 4, 5, 5])
{1, 2, 3, 4, 5}

>>> b = set([2, 3, 4, 5, 6])

>>> a.union(b)
{1, 2, 3, 4, 5, 6}

>>> a.intersection(b)
{2, 3, 4, 5}
```

Faster member checking

```
>>> a.difference(b)
{1}

>>> b.difference(a)
{6}

>>> a.symmetric_difference(b)
{1, 6}
```

LOOPS AND ITERATION

```
>>> for x in range(1,4):
        print('{0:3d} {1:3d} {2:3d}'.format(x, x*x, x*x*x))

  1   1    1
  2   4   8
  3   9  27
```

```
>>> x = 0
>>> while x < 3:
        print(x)
        x += 1

0
1
2
```

```
>>> a = [1, 2, „three”, 4, „five”]
>>> for i in a:
        if type(i) is str:
            print(i)

three
five
```

```
>>> string = „Blueberrymuffins”
>>> for i in string:
        if i == „b”:
            print(‚I found a (lowercase) b!’)

I found a (lowercase) b!
```

We can iterate over all basic datatypes, moreover, we can even define our own iterables (see later).

ENUMERATION AND ZIPPING

- Sometimes it is useful to know the index of the elements. There are two ways to do this in a loop:

```
>>> a = [1,4,23,8,4,9,11]
>>> for i in range(len(a)):
        print(i, a[i])
>>> for i, elem in enumerate(a):
        print(i, elem)
```

- The other way around, when you have two (or more) lists with the same size, and you want to iterate over them in a single loop (without the range iterator):

```
>>> a = [1,4,23,8,4,9,11]
>>> b = [7,1,4,3,121,137,42]
>>> for elem_a, elem_b in zip(a, b):
        print(elem_a, elem_b)
```

The enumerate and zip functions work for every iterable type.

LIST COMPREHENSION

Now that we wrote loops, we can define lists easier and based on more complex rules.

This is called list comprehension.

```
>>> a = [x*x  for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
>>> a = [x*x for x in range(10) if x%2 == 0]
[0, 4, 16, 36, 64]
```

```
>>> a = [i if i % 2 == 0 else 2*i for i in
range(10)]
[0, 2, 2, 6, 4, 10, 6, 14, 8, 18]
```

```
>>> b = [[1,2,3],[4,5,6],[7,8,9]]
```

```
>>> a = [i for x in b for i in x]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

LIST COPYING

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

>>> b = a

>>> b[3] = 5

>>> print(a)
[1,2,3,5]
```

Ooops! Not what we wanted!

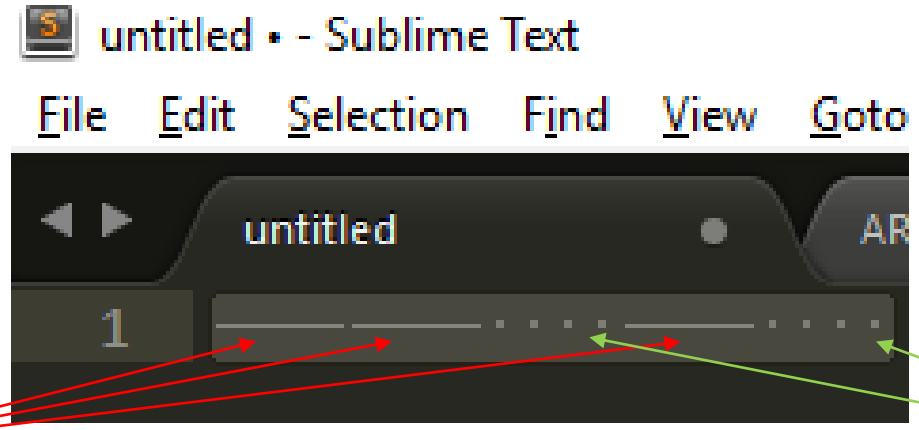
```
>>> b = a[:]  # (or a.copy())

>>> b[3] = 4

>>> b
[1,2,3,4]

>>> a
[1,2,3,5]
```

IDENTATION AND ENCODING

- We are leaving the python prompt now! WARNING!
- Python uses whitespace indentation to delimit code blocks! (4 white space is recommended).
- The standard is called PEP-8. Follow it!
- Choose (or set) your editor wisely!
- Use UTF-8 encoding!

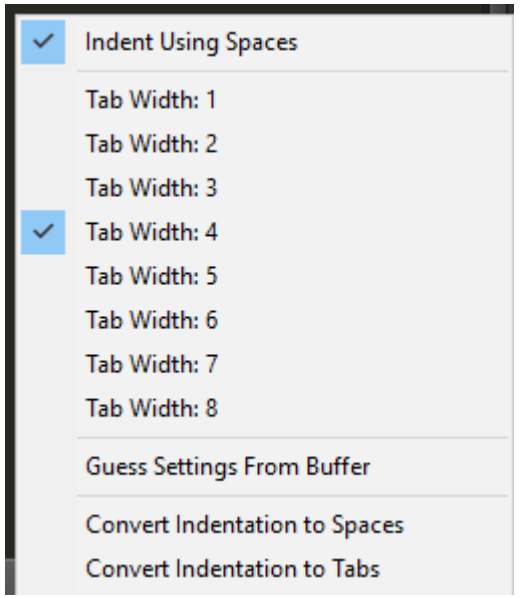
TIPS FOR EDITOR

- DON'T MIX tabs and whitespaces! (The code won't run and gives you hours of headaches!) Tip: Sublime Text 3 (freeish, multiplatform) and not using tab characters.
- When highlight text, shows tabs and whitespaces with different signs (dashes and dots).
- Can be enhanced with various Python-related extensions (style checking, autocompletion etc.)
- Atom, vim, emacs, Spyder, PyCharm etc.



Tabs

Four whitespaces



DEFINING FUNCTIONS

```
def my_func(a, b, c=2, d=0):
        """This is a documentation string."""
        return a+b+c+d  # And this is a
comment.

>>> my_func(1, 2)
5

>>> args = (1, 2)
```

```
>>> kwargs = {‚c’: 3, ‚d’: 4}

>>> my_func(*args, **kwargs)
10

>>> help(my_func)
Help on function my_func in module
__main__:
my_func(a, b)
    This is a documentation string.
```

HELP AND DIR

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

>>> help(a)
Help on list object:

class list(object)
 |  list() -> new empty list
 |  list(iterable) -> new list initialized
from iterable's items
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
[…]
```

```
>>> dir(a)
[[…], 'append', 'clear', 'copy', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

>>> ['__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', 'a',
'args', 'kwargs', 'my_func']
```

SPECIAL FUNCTIONS – GENERATORS

- The simplest form of generators are functions appended with the yield statement.
- Purpose: feed iterations. BUT! Can be used for wild things.

```
def countdown(n):
    print("Counting down from {}.".format(n))
        while n > 0:
            yield n
            n -= 1
```

```
>>> a = countdown(5)
>> next(a)
Counting down from 5.
5
>> for i in a:
            print(i)
4 3 2 1
```

SPECIAL FUNCTIONS – GENERATORS

- Remembers the state from before.
- Faster, than a list (if iterated).
- Lazy evaluation.
- Don’t have to implement the iterator protocol.
- Memory efficient.
- Once consumed, can’t be reused.
- Processing pipelines can be generated.
- Infinite sequences can be generated.

DEFINING CLASSES

```
class Spam(object):
    a = 1

    def __init__(self, b):
        self.b = b

    def imethod(self):
        pass

    @classmethod
    def cmethod(cls):
        return a

    @staticmethod
    def smethod():
        return 3.141592
```

Usage:

```
s = Spam(2)
s.imethod()
Spam.cmethod()
Spam.smethod()
```

- **Instance method:**
You have to instantiate the class.
- **Classmethod:**
You can call it without instantiating the class.
- **Static method:**
Just a function in the class, regardless of the class' purpose.

WRITING ITERATORS

```
class Counter(object):
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def __next__(self):
        if self.current >= self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1
```

Iterators are classes, that have to implement the iterator protocol (__iter__ and __next__ functions).

```
>>> a = Counter(0,10)
>>> for i in a:
        print(i, end=„ „)

0 1 2 3 4 5 6 7 8 9
```

MAGIC METHODS IN CLASSES

- We used __init__, __iter__ and __next__ so far.
- We can use them for operator overloading and redefining default methods.

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__']
```

CLASS INHERITANCE

```
class Base(object):
    def spam(self):
        print(„Base”)

class Foo(Base):
    def spam(self):
        print(„Foo”)
        # Call method in base class
        super().spam()
```

```
>>> a = Base()

>>> b = Foo()

>>> a.spam()
„Base”

>>> b.spam()
„Foo”
„Base”
```

Calling the base class' method from the derived class.

DIRECTORIES

Almost every object in Python is represented by dictionaries (under the hood)

```
class Spam:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def foo(self):
        print(„Foo”)
```

```
>>> s = Spam(2, 3)

>>> s.__dict__
{‚y’: 3, ‚x’: 2}

>>> Spam.__dict__
mappingproxy({'foo': <function Spam.foo at
0x03EB11E0>, '__weakref__': <attribute
'__weakref__' of 'Spam' objects>, '__dict__':
<attribute '__dict__' of 'Spam' objects>,
'__init__': <function Spam.__init__ at
0x03EB1198>, '__doc__': None,
'__module__': '__main__'})
```

Makes the dict members read-only (except the setattr), and assures, that all attributes and methods are strings.

Let’s taste metaprogramming

REPETITION

DRY

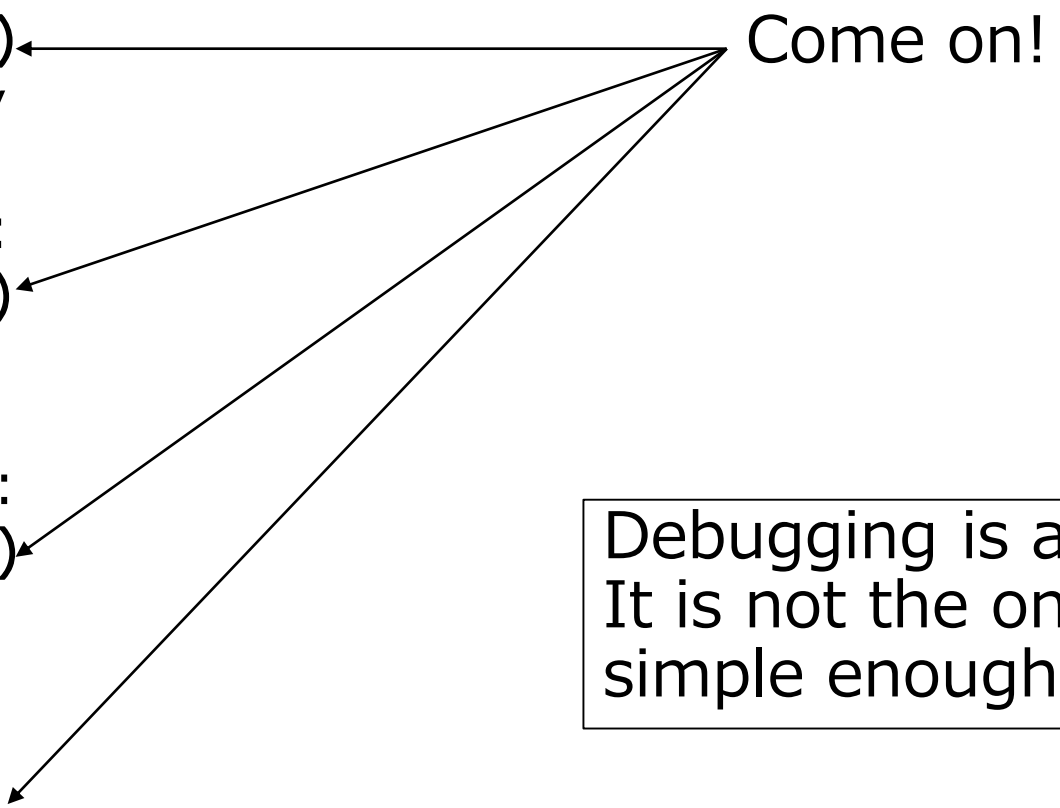
Don’t Repeat Yourself

- Repetitive code sucks.
- Boring to write.
- Hard to read.
- Hard to modify.

EXAMPLE: DEBUGGING

Debugging with print:

```
def add(x, y):
    print('add')
    return x+y

def sub(x, y):
    print('sub')
    return x-y

def mul(x, y):
    print('mul')
    return x*y

def div(x, y):
    print('div')
    return x/y
```



Debugging is a good example.
It is not the only example, but it is simple enough to fit on the slides.

EXAMPLE: DEBUGGING

- Decorators are functions, which creates a wrapper around the input function.
- A wrapper is a new function, which behaves exactly like the old function, takes the same arguments, but it can do further operations compared to the original function.

DECORATORS

```
from functools import wraps

def debug(func):
    msg = func.__qualname__

    @wraps(func)
    def wrapper(*args, **kwargs):
        print(msg)
        return func(*args, **kwargs)
    return wrapper
Application:
>>> func = debug(func)
```

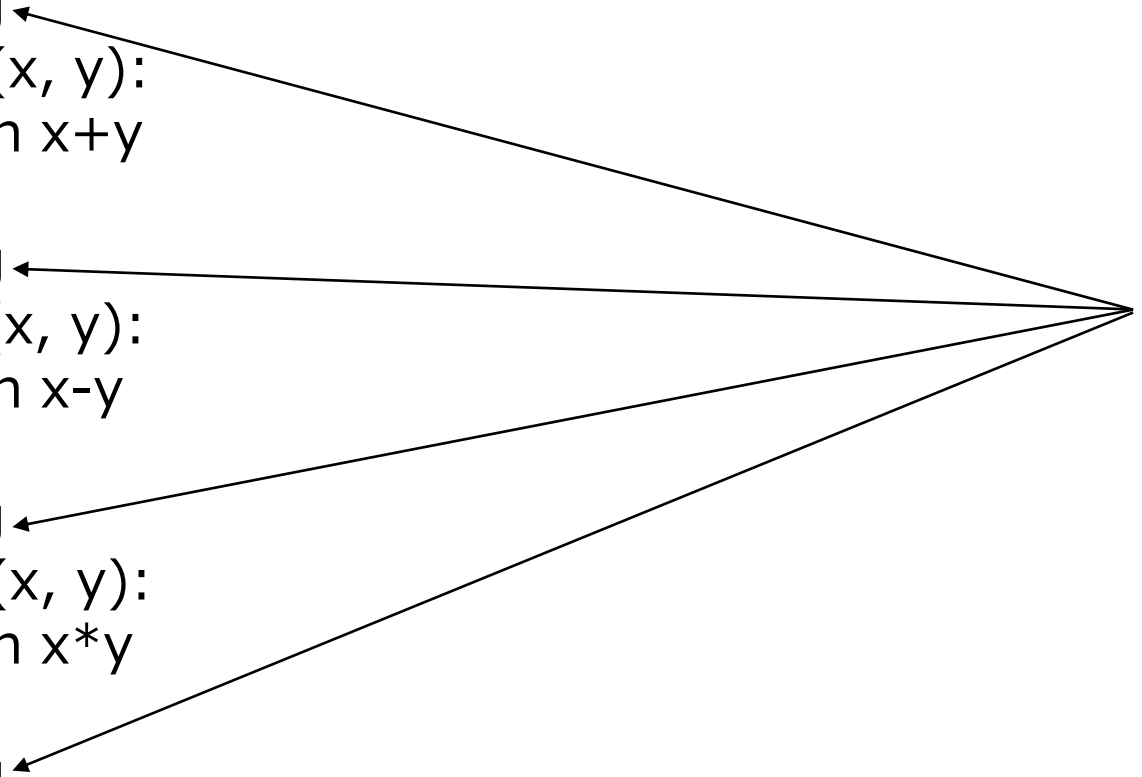
The decorator creates a wrapper function around the function.

Copies the metadata (like arguments)

DECORATORS

Alternative usage (this it the common one):

```
@debug
def add(x, y):
    return x+y

@debug
def sub(x, y):
    return x-y

@debug
def mul(x, y):
    return x*y

@debug
def div(x, y):
    return x/y
```



We decorated all the functions, but we don't see the details of the implementation.

WHY DEBUG THIS WAY?

- We separated the debug and the original code.
- We only had to write it once.
- Easier to change (or switch on and off).
- The user of the decorator don’t have to worry about the underlying implementation.

WRAPPING CAN BE OPTIONAL

Important: the decorator can be modified regardless of the code using it!

```
from functools import wraps
import os

def debug(func):
    if ‚DEBUG’ not in os.environ:
        return func


    msg = func.__qualname__


    @wraps(func)
    def wrapper(*args, **kwargs):
        print(msg)
        return func(*args, **kwargs)
    return wrapper
```

CLASS DECORATORS

Idea:

- Walk through the dictionary of the class.
- Determine the callable methods (e.g. functions).
- Wrap this with a decorator.

CLASS DECORATORS

```
def debugmethods(cls):
    for name, val in vars(cls).items():
        if callable(val):
            setattr(cls, name, debug(val))
        return cls
```

```
@debugmethods
class Spam:
    def grok(self):
        pass

    def bar(self):
        pass

    def foo(self):
        pass
```

CLASS DECORATORS

- We apply the decorator only once.
- Decorates all the functions of the class.
- Mostly it works...
- Class methods and static methods are not wrapped!

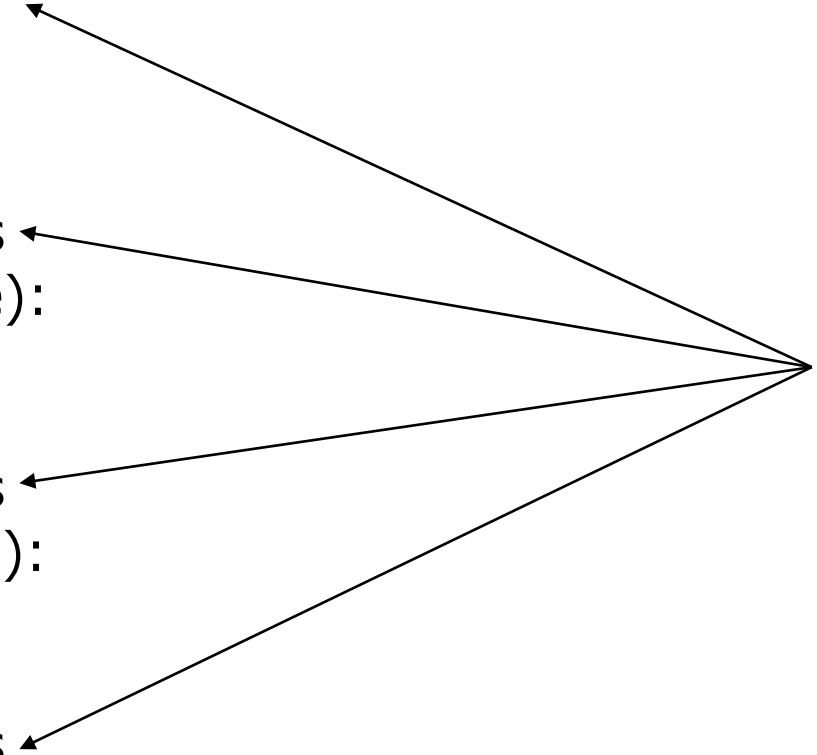
ANOTHER EXAMPLE

```
def debugattr(cls):
    orig_getattribute = cls.__getattribute__

    def __getattribute__(self, name):
        print(‚Get:’, name)
        return orig_getattribute(self, name)

    cls.__getattribute__ = __getattribute__

    return cls
```

```
@debugattr
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

>>> p = Point(2, 3)
>>> p.x

Get: x
2

>>> p.y
Get: y
3
```

DEBUGGING MULTIPLE CLASSES

```
@debugmethods
class Base:
    …

@debugmethods
class Spam(Base):
    …

@debugmethods
class Grok(Spam):
    …

@debugmethods
class Mondo(Grok):
    …
```



Not again!!! Many classes with class decorators. Isn't it familiar? Didn't we already solve this?

SOLUTION: METACLASSES

```
class debugmeta(type):
    def __new__(cls, clsname, bases, clsdict):
        clsobj = super().__new__(cls, clsname, bases, clsdict)
        clsobj = debugmethods(clsobj)
        return clsobj
```

The class is created normally.

And immediately wrapped.

Usage:

```
class Base(metaclass=debugmeta):
    ...

class Spam(Base):
    ...
```

DERIVED CLASSES

The metaclass will propagate through the whole class hierarchy.

```
class Base(metaclass=debugmeta):
    ...

class Spam(Base):        # metaclass = debugmeta
    ...

class Grok(Spam):        # metaclass = debugmeta
    ...
```

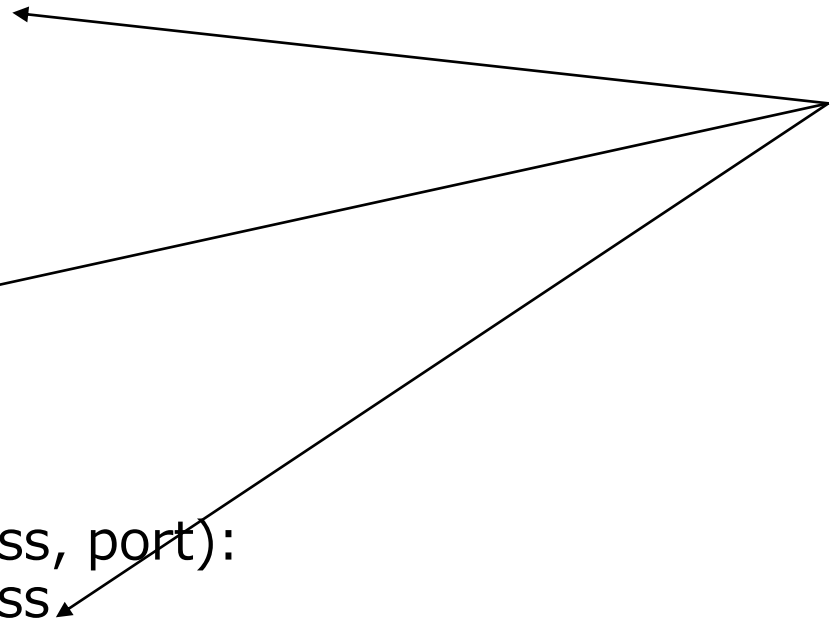
CLASS HIERARCHY

Wrapping:

- Decorators: functions
- Class decorators: classes
- Metaclasses: class hierarchies

EXAMPLE, WHICH IS NOT DEBUGGING

```
class Stock:
    def __init__(self, name, shares,
price):
        self.name = name
        self.shares = shares
        self.price = prices

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Host:
    def __init__(self, address, port):
        self.address = address
        self.port = port
```



Why do we have to write them over and over again?

SOLUTION: INHERITANCE

```
class Structure:
    _fields = []
    def __init__(self, *args):
        if len(args) != len(self._fields):
            raise TypeError(‚Wrong # args’)
        for name, val in zip(self._fields, args):
            setattr(self, name, val)

class Stock(Structure):
    _fields = [‚name’, ‚shares’, ‚price’]

class Point(Structure):
    _fields = [‚x’, ‚y’]

class Host(Structure):
    _fields = [‚address’, ‚port’]
```

SOLUTION: INHERITANCE

```
>>> s = Stock(‚ACME’, 50, 123.45)

>>> s.name
‚ACME’

>>> s.shares
50

>>> s.price
123.45
```

```
>>> p = Point(4, 5)

>>> p.x
4

>>> p.y
5
```

METAPROGRAMMING SUMMARY

Metaprogramming is a method, with which the repetitive code or repetitive code structures can be:

- Reduced
- Separate the reduced code
- At debugging, the debug code can be switched on and off
- Reduction can be made on different levels:
  - Function
  - Class
  - Class hierarchy