

Using Python for Scientific Computing

Session 1 - Python Basics

Felix Steffenhagen

University of Freiburg

May 2, 2011



- 1 Overview
- 2 Why Python?
- 3 Installation
- 4 Documentation
- 5 Datatypes
 - Numerical Datatypes
 - Variables
 - Sequence Datatypes
 - Dictionaries & Sets
- 6 Control Flow Tools
 - Functions
 - conditional statements
 - Loops
 - List Comprehensions

- **Schedule:** Mo-Do, 9 AM - 1 PM
- **Material:** Slides and exercises available at:
<http://portal.uni-freiburg.de/cognition/Members/steffenhagen/pysci2011/>
- **Contact:** Felix Steffenhagen,
 - Email: felix@cognition.uni-freiburg.de
- **Language:** English/German?

Any questions? Don't hesitate to ask.

Monday

- Python Basics

Tuesday

- Functions, OOD, Data Management

Wednesday

- Scientific Libraries

Thursday

- Graphical User Interfaces

- More than just complex computations
- Shuffling data from and to different tools (e.g. databases, feeding other programs)
- Data conversion (text $\leftrightarrow$ numeric $\leftrightarrow$ visualization)
- Dealing with distributed data (databases, files, directories)
- Graphical User Interfaces
- Efficiency

Traditional Programming vs. Scripting

“A script is what you give the actors. A program is what you give the audience ”

(Larry Wall, Inventor of Perl)

Traditional

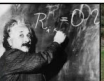
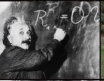
- Development of large software projects (e.g. browser, office)
- low abstract level
- System dependent (except Java)
- Usually very **fast** programs
- longer development time

Scripting

- Usually short programs for quick solutions. (e.g. data extraction)
- high abstract level, easier syntax
- Portable programs
- **slower!** But mostly acceptable.
- rapid development

- Developing programs on a higher abstraction level (than e.g. C/C++, Fortran, Java)
- Short programs to automate computer interaction
- Rapid prototyping, glue for software components
- Application Glue
- Why Python?
 - Platform-independent and object-oriented general-purpose language
 - easy and clean syntax (“Running Pseudo-Code”)
 - simple modularisation concept
 - Extensible and embeddable (e.g. extend with C/C++ $\Rightarrow$ *fast!*)
 - free and open-source + widely used and large community

How programmers see each other

Java	C	PHP	Ruby	Haskell	Lisp	as seen by...
						Java fans
						C fans
						PHP fans
						Ruby fans
						Haskell fans
						Lisp fans

Python is somewhere between Ruby and Lisp.

History of Python in a Nutshell

- Invented by Guido van Rossum in the late 80's at the *Centrum voor Wiskunde en Informatica* (Amsterdam)
- The name *Python* chosen for admiration of *Monty Python's Flying Circus*
- version 1.0 published in 1994,
- Current stable release: 2.7 / 3.2
- Open-Source community project, but important decisions still made by van Rossum, the *benevolent dictator for life* (BDFL).



Guido van Rossum

- Standalone programs or glue between applications
- Computer system administration, task automation
 - e.g. backup scripts
- Internet services and web applications
 - e.g. Zope, Plone, Django
- Scientific computing
 - Numeric computation
 - GIS
 - Life Sciences (Bioinformatics, Molecular Modeling)
- Embedded scripting language for applications (e.g. Gimp, Inkscape, ArcGIS)

Characteristics of Python

In short:

Python is an object-oriented programming language.

In more detail, Python is

- dynamically typed $\Rightarrow$ variables can have arbitrary types
- interpreted $\Rightarrow$ Need interpreter to run programs
- high-level $\Rightarrow$ High system abstraction
- object-oriented

Python includes an interactive interpreter. Very useful for testing code snippets.

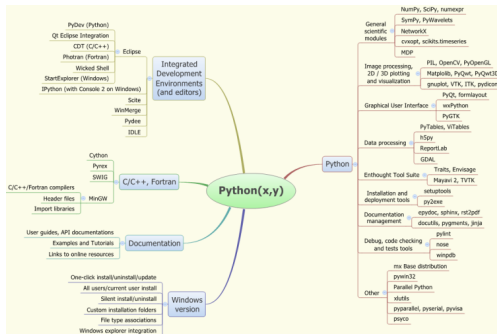
Many development utilities and cross-language support:

- Pure python: Python Interpreter + IDLE interpreter
- IPython: Enhanced interactive interpreter
- Pydev: Python support for Eclipse IDE
- Jython: Compiles Python code into Java programs
- IronPython: Compiles Python code into .NET programs
- Cython: Writing C/C++ extensions for Python
- python(x,y): Fully featured python distribution for scientific development



During this course we will work with python(x,y):

- Python Interpreter
- Enhanced interactive interpreter `ipython`
- Eclipse IDE
- scientific libraries (NumPy, SciPy, matplotlib)
- GUI-Libraries
- and a lot more ...



Installation

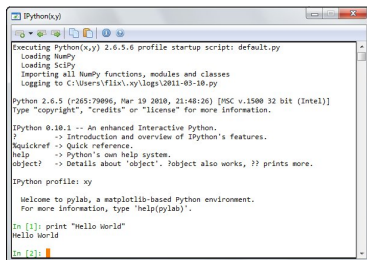
1 Getting python(x,y):

- Installation CD
- Download python(x,y) at <http://www.pythonxy.com>

2 Double-click and follow instructions

3 Click on

Start → Programs → Python(x,y) → Enhanced Consoles → IPython(x,y)



```
IPython(x,y)
Executing Python(x,y) 2.6.5.6 profile startup script: default.py
Loading NumPy
Loading SciPy
Importing all NumPy functions, modules and classes
Logging to C:\Users\Flix\xy\logs\2011-03-10.py

Python 2.6.5 (r265:79096, Mar 19 2010, 21:48:26) [MSC v.1500 32 bit (Intel)]
Type "copyright", "credits" or "license" for more information.

IPython 0.10.1 -- An enhanced Interactive Python.
? -> Introduction and overview of IPython's features.
Quickref -> Quick reference.
help -> Python's own help system.
object? -> Details about 'object'. ?object also works, ?? prints more.

IPython profile: xy

Welcome to pylab, a matplotlib-based Python environment.
For more information, type 'help(pylab)'.

In [1]: print "Hello World"
Hello World

In [2]:
```

Python Homepage: <http://www.python.org>

- Python Tutorial
<http://docs.python.org/tutorial/>
- Library Reference
<http://docs.python.org/library/>
- Module Index
<http://docs.python.org/modindex.html>

- Swaroop C. H.: **A Byte of Python** (english / german)
<http://www.swaroopch.com/notes/Python>
- Mark Pilgrim: **Dive into Python** (english)
<http://www.diveintopython.org>
- Elkner, Downey, Meyers: **How to think like a Computer Scientist** (english)
<http://openbookproject.net/thinkcs/python/english2e/>
- Kaiser, Ernesti: **Python - Das umfassende Handbuch** (german)
<http://openbook.galileocomputing.de/python/>

Running Python Programs

Python programs can be written to a file and executed using the Interpreter

hw.py

```
# This is a comment  
print "Hello World"
```

```
sh:> python hw.py  
Hello World
```

or we can call python commands in the interactive session

Interactive Session

```
>>> print "Hello World"  
Hello World
```

Simple output

For simple code examples we will use the interactive interpreter.

Two ways to output values in the interpreter:

Using print

```
>>> print 7 * 6
42
>>> print 'Hello, ' + 'World'
Hello, World
>>> print 'spam' * 3
spamspamspam
```

Direct output

```
>>> 7 * 6
42
>>> 'Hello, ' + 'World'
'Hello, World'
>>> 'spam' * 3
'spamspamspam'
```

`print` offers a lot more functionality → more of that later

All you do in programming is working with data. Mastering the available data types is a first step of mastering programming.

- Datatypes are the building blocks of programs
- Concept-specific storage of data
- Functions for creation and manipulation
- Python comes with very handy and powerful built-in datatypes
 - Boolean: True/False
 - Numerical: int, long, float, complex
 - Sequences: String, List, Tuple
 - Dictionaries, Sets

Numerical Datatypes

- Numerical types available in Python:
 - `int` for integer numbers -2147483648 .. 2147483647¹
e.g. -123, 23, 42
 - long integer numbers of **arbitrary** length
e.g. -123L, 23L, 9876543210L
 - `float` floating point numbers
e.g. -12.3, 2.3, 4.2e+6
 - complex complex floating point numbers
e.g. (4 + 2j), 2j, 3.1 + 1.3j

The difference between `int` and `long` can be forgotten because of automatic type conversion.

```
>>> 12345 * 12345
152399025
>>> 1234567890 * 9876543210
12193263111263526900L
```

¹more on a 64Bit computer

Operations on numerical values (int, long)

Basic operations on int, long

```
# Basic computation, Multiple values separated by ','
>>> print 69+42, 69-42, 69*42, 69/42
111 27 2898 1
# Force float division (Python <3.0)
>>> print float(69) / 42, 69.0 / 42
1.64285714286 1.64285714286
# Modulo operator (Rest of integer division)
>>> 69 % 42
27
# Power operator **
>>> print 2**2, 3**2, 2**32
4 9 4294967296
```

Operations on numerical values (float, complex)

Basic operations on float, complex

```
# Basic computation
>>> print 6.9 + 4.2, 6.9 - 4.2, 6.9 * 4.2, 6.9 / 4.2
11.1 2.7 28.98 1.64285714286

# exponential notation
>>> print 1.234e+6 + 999
1234999.0

# Complex numbers
>>> 2+3j + 4-1j
(6+2j)

# multiplication first!
>>> print 4+2j * 100, (4+2j) * 100
(4+200j), (400+200j)
```

When combining different numerical types in operations, Python automatically converts the result to the most general (best fitting) type.

Variables

We also want to store values in variables for further use. This works very intuitively in Python.

Variable assignments

```
>>> spam = 42 * 7 * 6
>>> print spam
1764
# Variables can hold any data type
>>> spam = 'spam, spam, spam'
>>> print spam
spam, spam, spam
# Variables must be declared before calling
>>> print 22 + egg
Traceback (most recent call last):
  File '<stdin>' line 1, in <module>
NameError: name 'egg' is not defined
```

Multiple assignments (Tuple Unpacking)

It is possible to assign multiple value at once:

Multiple Assignments

```
>>> spam, egg = 17, 4
>>> print 'spam =', spam, ', egg =', egg
spam = 17 , egg = 4
```

All values will be evaluated **before** assignment.

Swap variables without temporary variable

```
>>> spam = 'first'
>>> egg = 'second'
>>> spam, egg = egg, spam
>>> print spam, egg
second first
```

Sequence Datatypes

Sequence datatypes hold elements of other datatypes in a specific order.

- Strings (character sequences), Lists, Tuples

Python Interpreter

```
>>> first_name = "Homer"
>>> last_name = "Simpson"
>>> name = first_name + " " + last_name
>>> print name
Homer Simpson
>>> data = [15.79, 22.84, 11.7]
>>> print data[1], sum(data)
22.84 50.33
>>> primes = (2, 3, 5, 7, 11)
>>> print primes[1:4]
(3, 5, 7)
```

- Python distinguishes ASCII byte strings (standard encoding) and Unicode strings
- For platform independency and the usage of special characters you should specify an encoding.

german.py

```
# -*- coding: utf-8 -*-  
print("Münchener Hofbräuhaus")
```

- Tells Python to use UTF-8 encoding which allows usage of special characters
- Clean ASCII files do not need a coding specification

String Literals

- There are several ways to define strings

String definitions

```
>>> print "Double quoted string"
Double quoted string
>>> print 'Single quoted string'
Single quoted string
>>> print "A string with \"quotes\""
A string with "quotes"
>>> print """Now we can use "quotes" inside."""
Now we can use "quotes" inside.
>>> print "First Line\nSecond Line"
First Line
Second Line
```

String Interpolation

- Replacement of special character sequences with variable content (similar to `printf` in C)
- Special sequences start with a `%`

Example

```
>>> name = "Homer Simpson"
>>> greeting = "Hello Mr. %s." % name
>>> print greeting
Hello Mr. Homer Simpson.
>>> x, y, z = 6, 7 , 6*7
>>> print "%s * %s = %s" % (x, y, z)
6 * 7 = 42
```

Special replacement operators

Most commonly used replacement operators:

- **%s**: Most often used replacement operator.
 - Converts the element into a string as if printed with `print`
- **%f** Used for floating point numbers.
 - The numbers decimal places can be specified by `.i`. The number will be rounded to `i` decimal places.

Python Interpreter

String interpolation

```
>>> pi, e = 3.14159, 2.71828
>>> print "PI=%s    e=%s" % (pi, e)
PI=3.14159    e=2.71828
>>> print "PI = %.2f" % pi
PI = 3.14
```

See the Python documentation for more operators.

Common string operations

Let's have a look at some common string operations using the strings
`s = "Hello World"` and `n = "123654"`:

Function	Description	Result
<code>s.upper()</code>	convert to upper case	HELLO WORLD
<code>s.lower()</code>	convert lower case	hello world
<code>s.reverse()</code>	reverse string	dlroW olleH
<code>s.split()</code>	split string on whitespaces	["Hello", "World"]
<code>s.split('o')</code>	split on occurrence of 'o'	["Hell", " W", "rld"]
<code>s.replace(" ", ":")</code>	string replacement	Hello:World
<code>s.startswith("He")</code>	check beginning of string	True
<code>s.endswith("rld")</code>	check ending of string	True
<code>n.isalpha()</code>	consists of letters only?	False
<code>n.isalnum()</code>	string is alpha-numeric?	True
<code>n.isdigit()</code>	string is a number?	True

See Python documentation for more details and more operations on strings.

Lists and Tuples

- Tuples and lists are containers for other objects
- Tuples are defined with parentheses, lists with brackets
e.g. ("three", 2, "one") vs. ["alpha", "beta", "gamma"]
- Can contain arbitrary objects, even other tuples or lists
- Main difference:
 - lists are *mutable* (insert, append, remove)
 - tuples are *immutable*.

tuples and lists

```
coords = (1.4, 3.1, 2.7)
matrix = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
pairs = [ ("adam", "eve"), ("thelma", "louise") ]
```

Common list operations

Let's have a look at some common list operations using the lists $l = [1, 2, 3]$ and $m = ["o", "t"]$

Function	Description	Result
<code>l.append(42)</code>	append an element	<code>[1, 2, 3, 42]</code>
<code>l.extend(m)</code>	extend list with another list	<code>[1, 2, 3, "o", "t"]</code>
<code>l.insert(0, "zero")</code>	insert element at index	<code>["zero", 1, 2, 3]</code>
<code>l.pop()</code>	remove and return last item	<code>3</code> # $l = [1, 2]$
<code>l.index(2)</code>	find index of value	<code>1</code>
<code>l.sort()</code>	sortiert die Liste	<code>None</code> # $l = [1, 2, 3]$
<code>l.reverse()</code>	reverse the list	<code>None</code> # $l = [3, 2, 1]$

See Python documentation for more details and more operations on lists.

Operations on sequences

All sequence types (`str`, `list`, `tuple`) support the following operations:

- **Concatenation:** `"cheese" + "cake" == "cheesecake"`
- **Repetition:** `[0] * 3 == [0, 0, 0]`
- **Indexing:** `"GATTACA"[4] == "A"`
- **Member tests:** `9 in [2, 4, 9, 16] ⇒ True`
- **Slicing:** `"[Alpha Beta Gamma][6:10] == "Beta"`
- **Iteration:** `for i in [5, 4, 3, 2, 1]`

Concatenation and Repetition

```
# Put stars around spam
>>> s = "spam" * 3           # spamspamspam
>>> stars = "*" * 5          # *****
>>> print stars + s + stars
*****spamspamspam*****

# Combining two lists
>>> zeros = [0] * 4           # [0, 0, 0, 0]
>>> ones  = [1] * 4           # [1, 1, 1, 1]
>>> print zeros + ones
[0, 0, 0, 0, 1, 1, 1, 1]

# Lists with heterogenous content
>>> print ["one", 2, 3, 4] * 2
['one', 2, 3, 4, 'one', 2, 3, 4]
```

- First element of a sequence always at index 0
- Sequences can be indexed from beginning or end
- Use negative indexes to start at indexing at the end, last element at index -1

Indexing

```
>>> primes = [2, 3, 5, 7, 11, 13, 17]
>>> print primes[0], primes[1], primes[-1]
2 3 17
# Strings are also sequences
>>> print "Python"[2]
't'
>>> print "Python"[10]
IndexError: string index out of range
```

Member tests: the `in` operator

- For tuples and lists:
`item in seq` returns `True`, if `item` is an element of `seq`
- For strings
`substr in mystring` returns `True` if `substr` is a substring of `mystring`

Member tests

```
>>> print 5 in [1, 3, 5, 7]
True
>>> print "CAT" in "GATC", "pam" in "spam", "m" in "spam"
False True True
>>> menu = ("pizza", "pasta", "spam", "donuts")
>>> if "donuts" in menu:
>>>     print "hmmm ... donuts"
hmm ... donuts
```

Slicing is the extraction of subparts (slices) of the original sequence.

Slicing

```
>>> numbers = range(10)           # [0, 1, 2, 3, 4, 5, 6, 7, 8 ,9]
>>> print numbers[3:6]             # last index not included!
[3, 4, 5]
>>> print numbers[:5]              # all from start to 5th
[0, 1, 2, 3, 4]
>>> print numbers[6:]              # all from 6th
[6, 7, 8, 9]
>>> print numbers[5:-2]            # all from 5th to 2nd to last
[5, 6, 7]
```

Slicing in Detail

- `seq[i:j]` returns the part `[i, )`, i.e. the elements at positions `i, i+1, i+2, ..., j-1`
- leaving out `i` $\Rightarrow$ start from position 0:
`("one", "two", 3, 4)[1:3] == ("two", 3)`
- leaving out `j` $\Rightarrow$ end at last position:
`("one", "two", 3, 4)[1:] == ("two", 3, 4)`
- leaving out both $\Rightarrow$ **copy the whole sequence**
`("one", "two", 3, 4)[:]` $==$ `("one", "two", 3, 4)`

There are no `IndexErrors` in slices

```
>>> print "Python"[2:10], "ring"[-8:3]
'thon' 'rin'
>>> "spam"[10:]
[]
```

- You can also set a stepsize for skipping indices:

Extended Slicing

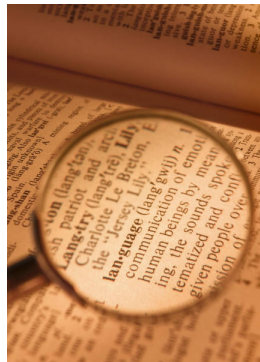
```
>>> numbers = range[10]      # [0, 1, 2, ..., 9]
>>> numbers[1:7:2]           # from 1 to 7 in steps 2
[1, 3, 5]
>>> numbers[1:8:2]           # from 1 to 8 in steps 2
[1, 3, 5, 7]
>>> numbers[7:2:-1]          # from 7 backwards
[7, 6, 5, 4, 3]
>>> numbers[::-1]            # full reverse copy
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Dictionaries

- Sequence datatypes ordered by a numerical index.
- Dictionaries (aka. associative arrays, hash maps) are lookup tables that are indexed by arbitrary^a keys instead of numbers^b
- No specific order! Memory positions are computed from keys.
- Fast direct element access.
- Mostly used as data container for direct access instead of iteration like lists.

^aWell, there are some constraints.

^bBut numbers are also possible.



Dictionary examples

```
>>> eng2ger = {"hello":"hallo", "world":"welt",
...           (1,2,3):"Eins, Zwei, Drei"}
>>> print eng2ger["hello"]      # Access
hallo
>>> eng2ger["hello"] = "hi"      # Update
>>> eng2ger["new"] = "neu"       # Insert
>>> print eng2ger
{'world': 'world', 'hello': 'hi', 'new': 'neu',
(1, 2, 3): 'Eins, Zwei, Drei'}
```

Note the element order when printing the dictionary!

Creating Dictionaries

Dictionaries can be created using several methods:

- `{key1:value1, key2:value2, ... }`
keys and values are normal Python objects (e.g. strings, numbers, tuples)
- `dict(key1=value1, key2=value2, ...)`
keys are variable names that are converted into strings, values are normal Python objects
- `dict(pair_sequence)`
e.g. `dict([(key1, value1), (key2, value2), ...])`.

Three ways for creating the same dict

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

There are more ways but this is sufficient for now.

Accessing Dictionaries

- `d[key]`:
Returns the value for element with key `key`. `KeyError`, if `key` not in dictionary.
- `d.get(key[, default])`: Like `d[key]` but in case of `KeyError` return `default` or `None`, if not default value was specified.

Examples

```
>>> coords = {"Freiburg":(47.9, 7.8), "Berlin":(52.5, 13.4)}
>>> print coords["Freiburg"]
(47.89, 7.85)
>>> print coords["Vancouver"]
KeyError: 'Vancouver'
>>> print coords.get("Vancouver", (0.0, 0.0))
(0.0, 0.0)
```

Updating Dictionaries

- `d[key] = value`
Inserts or updates element `key` with `value`
- `d.update(another_dict)`
Transfers all values of `another_dict` to `d`. Existing entries will be overwritten.
- `d.update` accepts all arguments like for creating a dictionary (see dict creation)

Examples

```
>>> coords = {"Freiburg":(47.8, 7.8), "Berlin":(52.5, 13.4)}
>>> c_new = {"Freiburg":(47.98, 7.85), "Oslo":(59.9, 10.7)}
>>> coords["London"] = (51.5, -0.1)
>>> coords.update(c_new)
>>> print coords
{'Freiburg': (47.98, 7.85), 'Oslo': (59.9, 10.7),
'Berlin': (52.5, 13.4), 'London': (51.5, -0.1)}
```

Deleting Elements from a Dictionary

- `del d[key]` Removes element with key `key`.
- `d.pop(key[, default])` Removes and returns element with key `key`.
- `d.clear()` Removes all elements from `d`.

Examples

```
>>> d = { "one":1, "two":2, "three":3 }
>>> d.pop("two")
2
>>> d
{'one': 1}
>>> d.clear()
>>> d
{}
```

See Python documentation for more details and more operations.

- Since Python 2.4 `set` is a builtin datatype.
- Sets are *unordered* collections of elements.
- Elements in sets are *unique*; A set can not contain the same element more than once.
- As for dictionaries: Elements must be *hashable* (immutable)
- `set` vs. `frozensets`:
 - `frozenset` is immutable $\Rightarrow$ hashable
 - `set` is mutable
 - `frozensets` therefore can be elements of other sets.

- `set()`, `frozenset()`:
Creates an empty set/frozenset
- `set(seq)`, `frozenset(seq)`:
Creates set/frozenset from given sequence

Python Interpreter

```
>>> print set("spamspam")
set(['a', 'p', 's', 'm'])
>>> print set(["spam", 1, [2, 3]])
TypeError: list objects are unhashable
>>> print set(["spam", 1, (2, 3)])
set([1, (2, 3)], "spam")
```

Basic operations on Sets

Python Interpreter

```
>>> s = set([1,2,3,4,5])
>>> 3 in s                # Membership test
True
>>> "spam" in s           # Membership test
False
>>> bool(s)               # True if set is not empty
True
>>> len(s)                # Count number of elements
5
```

Sets also support iteration by loops (more on that later)

Adding and removing elements to sets

- `s.add(element):`
Add element to `s` if not in `s` already.
- `s.remove(element):`
Removes element from `s`. Error if element not in `s`.
- `s.discard(element):`
As `remove` but no error if element not in `s`.
- `s.pop():`
Returns a random element and removes it from `s`.
- `s.clear():`
Removes all elements

Set operations

Python Interpreter

```
>>> s, t = set(1,2,3,4), set(3,4,5,6)    # create two sample sets
>>> s.union(t)                            # union set
set([1, 2, 3, 4, 5, 6])
>>> s.intersection(t)                   # intersection
set([3, 4])
>>> s.difference(t)                     # s without t
set([1, 2])
```

Useful trick to make lists unique:

```
>>> numbers = [1, 1, 2, 2, 3, 3]
>>> numbers = list(set(numbers))
>>> print numbers
[1, 2, 3]
```

Functions

So far we can only write short expressions. For more complex programs we need functions.

Defining functions

```
>>> def product(x, y, z):  
...     return x * y * z  
...  
>>> print product(2, 3, 4)  
24
```

Characteristic Python Syntax:

- Code block structure is marked by **Indentations**
- Simple Rule: Everything that belongs to the function must be indented.
- Tabs and spaces are allowed for indentations (Don't mix!!!)

Structure of a function

```
>>> def triple(spam):  
...     return 3 * spam  
...
```

- Functions are started by the keyword `def`
- Accept a variable number of arguments (more on that later)
- Values are returned with the keyword `return`.
If no return value specified, function returns `None`
- Functions are polymorph, i.e. they do not check the parameter types
⇒ Type-checking has to be done by you!

Multiple return values

Functions can return multiple values at the same time:

Multiple return values

```
>>> def minmax(l):  
...     return min(l), max(l)      # return minimum and maximum  
...  
>>> print minmax([1,2,3,4])  
(1, 4)  
>>> a, b = minmax([1,2,3,4])  
>>> print a, b  
1 4
```

- Technically the function returns a tuple
- Assignments to several variables due to tuple unpacking
- More details on functions tomorrow.

if statements

So far we can not do much more than a basic calculator. For more complex program flow we need conditional branching.

Python Interpreter

```
>>> def factorial(n):  
...     if n <= 1:  
...         return 1  
...     else:  
...         return n * factorial(n-1)  
...  
>>> print factorial(5)  
120
```

The basic usage of `if` statements should be clear by this example.
Note the indentation!

General Structure of `if` statements

Beside `if` and `else` blocks, Python also supports `elif` blocks to construct more complex conditions.

General Structure of `if` statements

```
if <condition>:  
    <code>  
elif condition:  
    <code>  
elif condition:  
    <code>  
...  
else:  
    <code>
```

- `elif` and `else` branches are optional

Comparison operators

- For comparisons you can use operators as in Java or C++:
 - `==`, `!=`: Equality, Inequality
 - `<`, `<=`, `>`, `>=`: Ordinal comparisons
- Conjunction of conditions using `and`, `or` and `not`
- As in mathematics, comparisons can be chained:
 $0 \leq x \leq 10$ is equivalent to $(0 \leq x)$ **and** $(x \leq 10)$
- Any value/object can be used for comparisons. Types will be converted to boolean values.

Example: Complex conditional statement

Python Interpreter

```
def is_probably_prime(n):  
    if n == 2:  
        return True  
    elif n <= 1 or n % 2 == 0:  
        return False  
    elif n <= 20 and not (n == 9 or n == 15):  
        return True  
    else:  
        return n % 3 != 0 and n % 5 != 0
```

for loops

- Repeating code a certain number of times
- Python's `for` loop concept differs from other languages
- Most of the time `for` loops are used for iterating over items of a sequence
- Thus, `for` loops in Python always iterate over sequences

Python Interpreter

```
>>> primes = [2,3,5,7]
>>> product = 1
>>> for number in primes:
...     product = product * number
...
print product
210
```

The `range()` function

For common iteration with numerical progression you use the `range()` function to create a number sequence.

- `range()` creates lists of integer values
 - `range(stop)` creates `[0, 1, ..., stop - 1]`
 - `range(start, stop)` creates `start, start + 1, ..., stop - 1`
 - `range(start, stop, step)` creates `[start, start + step, start + 2 * step, ..., stop - 1]`
- `xrange()` works like `range` but creates an iterator object instead of a real list. Useful for pure looping to save memory.

Python Interpreter

```
>>> for x in range(10, 20, 2):  
...     print x  
#prints 10 12 14 16 18
```

Iterating over Sequences

Standard way of iterating over a sequence

```
for elem in s:  
    <Do Something>
```

- Sometimes you need to know the current index during iteration:
- `enumerate(seq)`: Creates an iterator that returns `(idx, value)` pairs

Using `enumerate` for iteration

```
for idx, elem in enumerate("egg"):  
    print "At position %s is a %s" % (idx, elem)  
At position 0 is a e  
At position 1 is a g  
At position 2 is a g
```

break, continue, else

When using loops, the following statements are of importance:

- `break` breaks out of the enclosing loop
- `continue` skips current step and continues with next iteration step
- `else:`
 - `for` loops can have an `else` branch.
 - executed when the loop is not terminated by a `break` statement

More complex for loop

Count letters until we find a number

```
count = 0
for char in "Python Yeah!42":
    if char.isdigit():
        break
    elif char.isalpha():
        count += 1
    else:          # not necessary, just for the show
        continue
else:
    print "No digit in the string"
```

List Comprehensions

- A common task when working with sequences is generating new lists from the containing elements.

- In math you use set comprehensions to describe sets:

$$\text{e.g. } S = \{3x \mid x \in A, x \geq 10\}$$

- Similar expressions are possible in Python using list comprehensions:

`S = [ 3*x for x in A if x >= 10]`

- Powerful and useful tool.
- Normally, you would use nested `for` loops and `if` statements
 - $\Rightarrow$ ugly indented code

```
>>> words = ["spam", "egg", "bacon", "beans", "cake"]
>>> print [ w.upper() for w in words if len(w) > 4 ]
['BACON', 'BEANS']
>>> print [ (x, x**2) for x in range(10) if x % 2 == 0 ]
[(0, 0), (2, 4), (4, 16), (6, 36), (8, 64)]
```

List Comprehensions: General Form

The general form of list comprehensions is as following:

```
[ <expression> for <var1> in <iterable1>
    for <var2> in <iterable2> ...
    if <condition1>
    if <condition2> ... ]
```

which is equivalent to

```
result = []
for <var1> in <iterable1>:
    for <var2> in <iterable2>:
        ...
        if <condition1>:
            if <condition2>:
                ...
                result.append(<expression>)
```

for loops vs. list comprehensions

Task: Generate squares of even numbers < 10

for loop vs list comprehension

```
squares = []  
for x in range(10):  
    if x % 2 == 0:  
        squares.append( (x, x**2) )  
  
# The same with list comprehension  
squares = [ (x, x**2) for x in range(10) if x % 2 == 0 ]
```

List comprehensions are intuitive and efficient for building subsets of lists.