

INTRODUCTION TO PYTHON

<http://www.liacs.nl/home/snijssen/CI/>

Python

- General-purpose
- Interpreted
- High-level
- Readable code
- Open source (CPython)

www.python.org

Python History

- Created and maintained by Guido van Rossum at the CWI (Amsterdam, 1980s), and at Google (California, now)
- Python 2.x is most common
- Python 3.x (released 2008) is the current standard
 - **not fully compatible with Python 2.x**

We will use what works best for us

Hello World

No header file, no main, no opening bracket

```
print "Hello world!"
```

No bracket

No ;

Variables

- Variables have types, but the type depends on what you assign to the variable
- Variables are not declared

```
a = 0  
b = "text"  
print a, b
```

↑
adds space as well

```
a, b = 0, "text"  
b, a = a, b  
print a, b
```

↑
simultaneous assignment
allowed

If-statements

No (...) C++ comparison : indicates start of block

```
a = 0
if a == 0:
    a = 1
    print "Zero"
elif a == 1:
    print "One"
else:
    print "Other"
```

Indentation indicates how long the block continues; no { ... }

else: if can be shortened

While-statement

Indentation
indicates
how long
the block
continues;
no { ... }

```
a = 0
while a < 10:
    print a
    a += 1
```

↑
a++ not supported

Functions

Defines function,
no return type

No parameter type

Indentation
indicates
how long
the block
continues;
no { ... }

```
def f(i):  
    return i + 1  
  
print f(1)
```


Classes

```
class Dimension:  
    width = 10  
    height = 10  
  
d = Dimension ()
```

Default value
for variable

Create instance by class + ()
(no new statement)

Reference semantics

```
class Dimension:
    width = 10
    height = 10

def f(dimension):
    dimension.width = 20

d1 = Dimension ()
d2 = d1 ←
f(d1)
print d1.width, d2.width ←
```

“all variables
are actually pointers”

objects are deleted
(garbage collection)
when there is no
pointer to them.

d2 points to the
same object as d1

prints “20 20”
(like Java)

Class constructors / methods

```
class Dimension:
    width = 10
    height = 10
    def __init__( self, w, h ):
        self.width = w
        self.height = h

    def write ( self ):
        print self.width, self.height

d = Dimension ( 3, 3 )
d.write ()
```

“constructor” is
always named
`__init__`

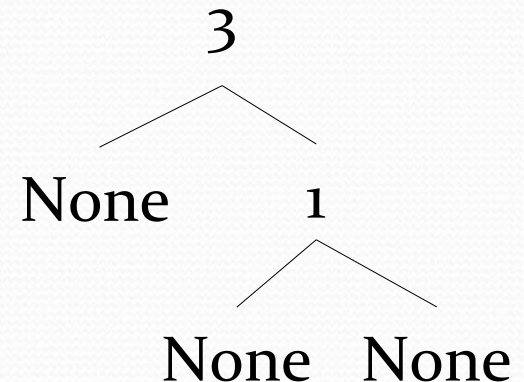
a “this” pointer
always needs to
be added
(and is called “self”)

nothing there

Classes: None

```
class Tree:  
    def __init__ ( self, v, l, r ):  
        self.left = l  
        self.right = r  
        self.value = v  
  
t = Tree ( 3, None, Tree ( 1, None, None ) )
```

↑
Instead of NULL



Operators

- Mathematical:

`*`, `+`, `-`, `%`, `&`, `|`, `~` : as in C++

`/` : division, always produces float

`//` : division, always produces integer

`**` : power-of

- Logical:

and : instead of `&&`

or : instead of `||`

not : instead of `!`

```
if a > 3 and b > 3:  
    print a, b
```

Arithmetic & Boolean Types

- Basic mathematical type names are: **float** (64 bit), **int** (32 bit), **long** (unlimited), **complex** (float real and imaginary)
 - conversions:

```
a = 3
b = float(a)
```
- Boolean: **bool**
 - however: **False** and **True** are written with capitals!

Lists

- Python has a built-in type for lists, and a syntax for constructing lists

```
a = [10, 20, 30]
b = [10, "something", 30 ]
```

Multiple types can be in the same list

- Lists are like arrays, but can do more

```
a = [10, 20, 30]
print a[0]
print a[-1]
```

Prints "10"
Prints "30"

(as also seen in logical and functional programming languages)

Lists

- Slices

```
a = [10, 20, 30]
print a[0:2]
print a[1:]
print a[:2]
```

Prints “[10, 20]”

- Length of a list

```
a = [10, 20, 30]
print len(a)
```

Lists

- Concatenation

```
a = [10, 20, 30]
b = a + [40, 50]
```

- Multiplication

```
a = [10, 20, 30] * 3 ←
```

[10, 20, 30,
10, 20, 30,
10, 20, 30]

- One can test for list membership

```
a = [10, 20, 30]
if 30 in a: print "in"
if 30 not in a: print "out"
```

(Be careful: uses
linear search)

Strings

- Strings are also lists

```
a = "text"  
print a[0:3], a*3
```

- Conversions to strings need to be done explicitly

```
a = 2  
b = "Nummer " + str(a)
```

- Many convenience functions for strings, eg.

```
print "--".join(["one", "two", "three"])
```



prints "one--two--three"

(More later)

Lists and for-loops

- Important functions that return lists: (Python 2.x)

`range(x)` → returns `[0,1,2,...,x-1]`

`range(x,y)` → returns `[x,x+1,...,y-1,y]`

- For loops are defined for lists (and iterators)

```
for i in range(4):  
    print i
```

Prints:

0

1

2

3

Sets

- Possible disadvantages of lists:
 - membership tests: linear search
 - elements can occur multiple times
- If problematic, use `sets`

```
a = { 2, 3, 3 }  
print a
```

use { } instead of []
prints "2 3"

```
a.add ( 1 )  
print a
```

```
a = set([10,20,30])  
print a
```


Tuples

- Tuples are *immutable* lists
 - immutable: the list cannot change (i.e., we can not add or remove a value in the list)

```
a = ( 1, 2 )  
print a
```

← use () instead of []

```
a = 1, 2  
print a
```

← also creates a tuple

```
a, b = 1, 2  
a, b = b, a  
print a, b
```

← “unpack” a tuple

← creates tuple for (b, a),
unpacks this in a, b

Dictionaries

- Dictionaries are like sets, but associate a *value* to each *key* in a set

```
a = { "anna" : 1, "bill" : 2 }  
  
for i in a:  
    print i, a[i]
```

: announces value

Prints:

bill 2

anna 1

→ only keys are retrieved in for

→ array-like notation to retrieve value

Dictionaries

- Updating dictionaries

```
a = { "anna" : 1, "bill" : 2 }  
a["christine"] = 3  
a.update ( { "donna" : 4, "eric" : 5 } )  
  
a["eric"] = 6  
  
a.pop ( "donna" )
```

adds
christine

Change value of "eric"

Remove "donna"

Reading files

```
f = open ( "test.txt" )  
for line in f:  
    print line  
f.close ()
```

open for reading

retrieve line-per-line
as if from a list

Note: `line` includes the end-of-line `\n`; after this `\n`, `print` by default puts another `\n`

Reading files

```
f = open ( "test.txt" )  
f.readline ()  
for line in f:  
    print line.rstrip ()  
f.close ()
```

Read one line

Remove white-
space on the
right (including
\n)

`line.lstrip ()`

Remove whitespace on the left

`line.strip ()`

Remove whitespace on the left and right

`line.split ()`

Splits line in words based on whitespace

Writing files

```
f = open ( "test2.txt", "w" )  
a = ["1\n", "2\n", "3\n"]  
f.writelines ( a )  
f.write ( "something" )  
f.close ( )
```

open for writing

to write a list, it must
consists of strings;
add \n for newlines

Modules & Pickle

```
import pickle ←  
  
a = [ [ 1, 2, 3 ], [ 2, 3, 4 ] ]  
  
f = open ( "dump", "w" )  
  
pickle.dump ( a, f ) ←
```

use the pickle
library

writes any standard
Python data
structure to disk

```
import pickle  
  
f = open ( "dump" )  
  
print pickle.load ( f )
```

Modules & Pickle

```
from pickle import dump ←  
a = [ [ 1, 2, 3 ], [ 2, 3, 4 ] ]  
f = open ( "dump", "w" )  
dump ( a, f ) ←
```

import one
function

no need to
add library
name

```
from pickle import * ←  
f = open ( "dump" )  
print load ( f )
```

import all
functions

Creating Modules

```
def increase ( x ):  
    return x + 1
```


mymodule.py

```
import mymodule ←  
print mymodule.increase ( 2 )
```

Looks in system
path and local
path for mymodule.py

Command line

```
import sys
print sys.argv
```



Contains a list of all command line arguments

or use the optparse module...

Other standard modules

- `math`
- `random`
- `gzip`
- `zipfile`
- `csv`
- `time`
- `optparse`
- `json`
- `xml`
- `...`

Exceptions

```
def search ( l, y ):  
    for x in l:  
        if x == y:  
            raise
```

Raises an exception

```
try:  
    search ( [2, 3, 1, 4], 3 )
```

Catch exceptions

```
except:  
    print "found"
```

Only executed if
exception raised

(note: `if 3 in [2,3,1,4]: print "found"`
would have been shorter)

Exceptions

Inherit from Exception class

```
class myException(Exception): pass
```

Empty class

```
def search ( l, y ):
```

```
    for x in l:
```

```
        if x == y:
```

```
            raise myException, y
```

Raises a specific exception

```
try:
```

```
    search ( [2, 3, 1, 4], 3 )
```

```
except myException, value:
```

```
    print value
```

Catch specific exception

Functions as Objects

- Python functions can be stored in variables

```
def f(i):  
    return i + 1  
  
a = f  
  
print a(1)
```

Function Closures

```
def add(i):  
    def sum(j):  
        return i + j  
    return sum
```

```
addone = add(1)
```

```
print addone(1)
```

value of `i` at the moment `g` is returned is stored together with `g` in a

Return function with one argument

Generators: yield

- if a function contains a `yield` statement, it can't have a `return` statement – when called, the function always immediately returns a generator object for itself
- each time the `next ()` operator is called, the function continues to be executed where it left off

```
def generator ( i ):  
    print i  
    yield  
    print i+1
```

```
a = generator ( 1 )  
a.next ( )  
a.next ( )
```

Generators: yield

- a yield statement can also “return” a value

```
def till ( n ):  
    i = 0  
    while i < n:  
        yield i  
        i += 1  
  
a = till ( 10 )  
print a.next ()  
print a.next ()
```

Generators & for loops

- for loops also apply to generators

```
def till ( n ):  
    i = 0  
    while i < n:  
        yield i  
        i += 1  
  
for i in till ( 10 ):  
    print i
```

- in Python 2.x, xrange(i) is a generator (range(i) returns a list)
- in Python 3.x range(i) is a generator (list(range(i)) creates a list by executing the generator)

List comprehension

```
a = [ i for i in xrange(10) if i != 2 ]  
print a
```

source of elements

condition(s) on elements

$\{i \in \{0, 1, \dots\} \mid i \neq 2\}$

Close to mathematical notation!

List comprehension

```
1. a = [ i for i in xrange(10) if i != 2 ]  
2. print a
```

```
1. vector<int> a;  
2. for ( int i = 0; i < 10; i++ )  
3.   if ( i != 2 )  
4.     a.push_back ( i );  
  
5. for ( int i = 0; i < 10; i++ )  
6.   cout << a[i] << " ";  
7. cout << endl;
```

List comprehension

```
def quicksort(list):
    if list:
        return \
            quicksort ( [ x for x in list[1:] if x < list[0] ] ) \
            + [list[0]] + \
            quicksort ( [ x for x in list[1:] if x > list[0] ] )
    else:
        return []

print quicksort([5,1,3,2,4])
```


Map

- Still long:

```
def f(i): return i+1
a = range(10)
b = [ f(i) for i in a ]
```

- Shorter:

```
def f(i): return i+1
a = range(10)
b = map(f, a)
```

↑
apply f on each element in the list

Map

```
a = [ [ 1, 2, 3 ], [ 2, 3, 4 ] ]  
  
f = open ( "output", "w" )  
  
for x in a:  
    f.write ( " ".join ( map ( str, x ) ) + "\n" )
```

Reduce

- Still long:

```
def f(i,j): return i+j
a = range(10)
b = 0
for i in a: b = f(i,b)
```

- Shorter:

```
def f(i,j): return i+j
a = range(10)
b = reduce(f,a)
```

Note: Google's map/reduce framework based on combining on a large scale map & reduce to perform calculations

Lambda functions

- Inline definition of functions without name

```
a = range(10)
b = reduce(lambda i,j: i+j, a)
```

lambda is a keyword;
function without
name

parameters of the
function

return
of the
function

Only useful for functions that can be written with one
expression

Filter

- Create sublist of a list based on boolean test

list empty?

closures work

```
def quicksort(list):
    if list:
        return \
            quicksort ( filter(lambda x:x < list[0], list[1:] ) ) \
            + [list[0]] + \
            quicksort ( filter(lambda x:x > list[0], list[1:] ) )
    else:
        return []

print quicksort([5,1,3,2,4])
```

Math-like notation

- Sum

```
print sum([i**2 for i in xrange(10)])
```

$$\sum_{i=0}^9 i^2$$

- Max

```
print max([i**2 for i in xrange(10)])
```

- And?

```
def land(l): return reduce(lambda x,y: x and y, l)
print land([True]*5)
```