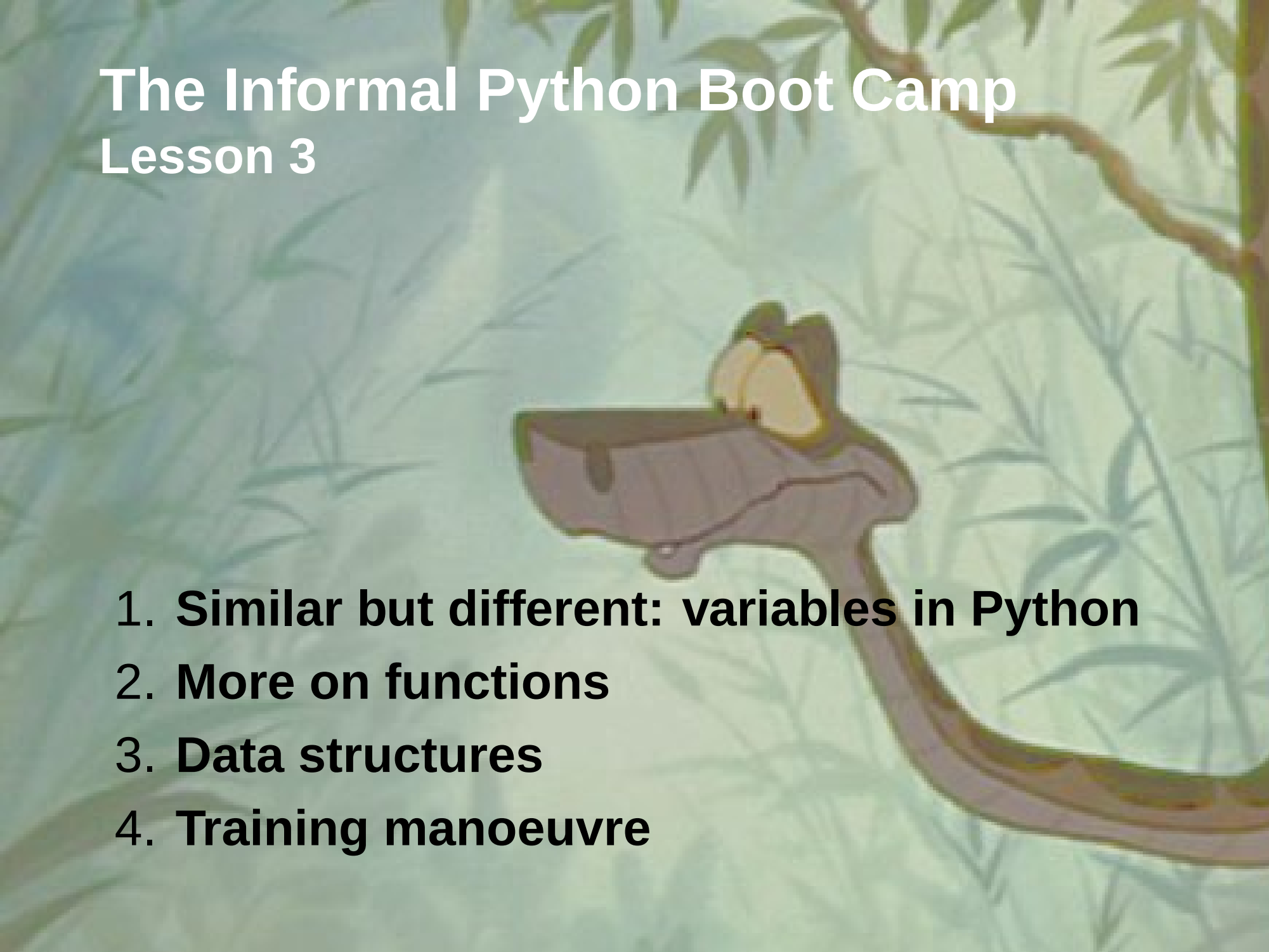
The Informal Python Boot Camp
Lesson 3



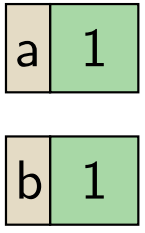
1. **Similar but different: variables in Python**
2. **More on functions**
3. **Data structures**
4. **Training manoeuvre**

Similar but different

Variables vs. identifiers

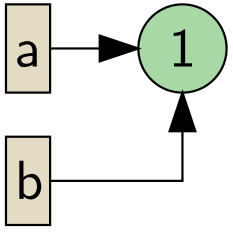
In C and Fortran, a variable behaves like a box containing a value:

```
a = 1;
b = a;
```



Python, in contrast, has names (or *identifiers*), which can be bound to objects:

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



Variables vs. identifiers

Example: add two numbers

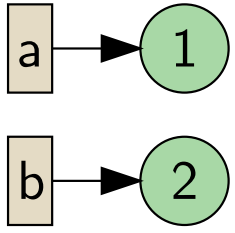
```
a = 1
b = 2
b = a + b
```

Let’s have a very close look at this primitiv operation...

Variables vs. identifiers

(step 1)

```
a = 1
b = 2
```

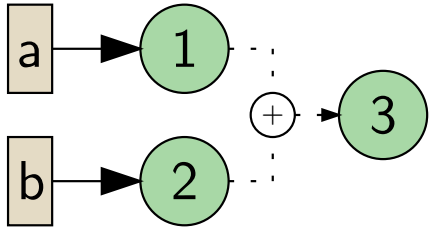


We start with two identifiers which are bound to two different number objects...

Variables vs. identifiers

(step 2)

```
a = 1
b = 2
    a + b
```

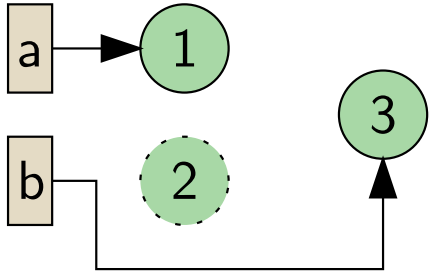


The + operator creates a new number object...

Variables vs. identifiers

(step 3)

```
a = 1
b = 2
b = a + b
```



Finally, the identifier `b` is rebound to the new number object.

Variables vs. identifiers

A consequence:

```
>>> a = [ 1 ]
>>> b = a           # b is now bound to the
                    # same list object as a.
>>> b.append( 2 )
>>> print a
[1, 2]              # !
```

More on functions

Arbitrary argument lists

Functions may take any number of additional arguments:

```
import os

def cautious_delete( prompt, *filenames ):
    for file in filenames:
        if 'y' == raw_input(prompt % file):
            os.remove(file)

cautious_delete( "Delete file '%s'? [y/N] ",
                 "lesson1.pdf", "lesson2.pdf" )
```

```
Delete file 'lesson1.pdf'? [y/N] y
Delete file 'lesson2.pdf'? [y/N] y
```

Arbitrary keyword arguments

Functions may also take additional keyword arguments:

```
def eat( **howmany_of ):
    for food in howmany_of.keys():
        print ("I will eat %i %s today!"
               % (howmany_of[food], food))

eat( apples=10, bananas=13, spagettis=333 )
```

```
I will eat 333 spagettis today!
I will eat 10 apples today!
I will eat 13 bananas today!
```

The keyword arguments are put in a *dictionary*.
More about *dictionaries* later.

Unpacking argument lists

Sometimes you have a function taking several arguments:

```
delete_files( 'lesson1.pdf', 'lesson2.pdf' )
```

but you have the intended arguments in a list:

```
files = [ 'lesson1.pdf', 'lesson2.pdf' ]
# doesn't work:
delete_files( files )
# doesn't always work:
delete_files( files[0], files[1] )
```

In these cases you can use *argument unpacking*:

```
delete_files( *files )
```

Documentation strings

```
def my_function():
    """Do nothing, but document it.

    No, really, it doesn't do anything.
    """
    pass

print my_function.__doc__
```

1. Brief summary (Start with capital letter, end with period.)
2. Blank line
3. Complete description

Input and output variables

Functions can return more than one value:

```
def spherical_coords( x, y, z ):
    # ...
    return r, theta, phi

r, theta, phi = sperical_coords( x, y, z )
```

Call by value (C)

In C we have *call by value*:

```
void f(int j) {
    j++;
    printf("inside: %i\n", j);
}

main() {
    int i;
    i = 1;
    printf("before: %i\n", i);
    f(i);
    printf("after:  %i\n", i);
}
```

```
before: 1
inside: 2
after:  1
```

Call by reference (C, with pointer)

In C we can use pointers to do a *call by reference*:

```
void f(int *j) {
    (*j)++;
    printf("inside: %i\n", *j);
}

main() {
    int i;
    i = 1;
    printf("before: %i\n", i);
    f(&i);
    printf("after:  %i\n", i);
}
```

```
before: 1
inside: 2
after:  2
```

Call by reference (C++)

C++ has built in an extra *call by reference*:

```
void f(int &j) {
  j++;
  cout << "inside: " << j << endl;
}

main() {
  int i;
  i = 1;
  cout << "before: " << i << endl;
  f(i);
  cout << "after:  " << i << endl;
}
```

```
before: 1
inside: 2
after:  2
```

Call by reference (Fortran)

Fortran always does *call by reference*:

```
subroutine f(j)
    integer :: j
    j = j+1
    print *, "inside:", j
end subroutine

program reference
    integer :: i = 1
    print *, "before:", i
    call f(i)
    print *, "after: ", i
end program
```

```
before: 1
inside: 2
after:  2
```

Call by object reference

Python, seems to do *call by value*...

```
def f(j):
    j += 1
    print "inside:", j

i = 1
print "before:", i
f(i)
print "after: ", i
```

```
before: 1
inside: 2
after:  1
```

... but this is not the case!

Inside the function, j is rebound to a new number object with the value 2. Identifier i is not rebound.

Call by object reference

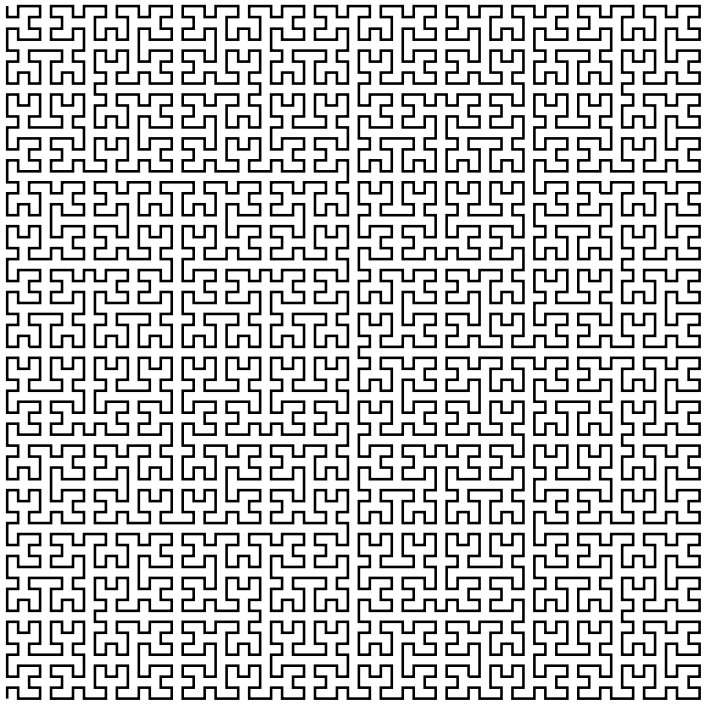
Python is passing *object references by value*.
Not the value of the object.

```
def f(j):
    j[0] += 1
    print "inside:", j[0]

i = [ 1 ]
print "before:", i[0]
f(i)
print "after: ", i[0]
```

```
before: 1
inside: 2
after:  2
```

Data structures



Lists - Tuples - Dictionaries

List methods to remember

- `append(x)`: Add `x` to the end.
- `extend(L)`: Append all elements of `L`.
- `insert(i,x)`: Insert `x` at position `i`.
- `pop()`: Remove and return last element.
- `pop(i)`: Remove and return element at position `i`.
- `remove(x)`: Remove first element whose value is `x`.
- `index()`: Return index of first element whose value is `x`.
- `sort()`: Sort list.
- `count(x)`: Return the number of times `x` appears.
- `reverse()`: Reverse order of elements.

Using lists as stacks

A list used as a *last in, first out* data structure:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack
[3, 4, 5]
```

Using lists as queues

A list used as a *first in, first out* data structure:

```
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry")       # Terry arrives
>>> queue.append("Graham")      # Graham arrives
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

List comprehensions

List comprehensions are used to create lists, based on the values of other lists:

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

Looping techniques

To enumerate items of a sequence:

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

Looping techniques

To iterate over several sequences simultaneously:

```
>>> for a,b,c in zip(['ene', 'raus'],
                     ['mene', 'bist'],
                     ['mu', 'du']):
...     print a, b, c
...
ene mene mu
raus bist du
```

Tuples

Tuples are immutable lists.

- Round parantheses create tuples.
- Parantheses can be ommited in many cases.

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Tuples

Creating tuples with zero or one element:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- trailing comma!
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

Dictionaries

Dictionaries are used to store *key: value* pairs.

- Braces create dictionaries.
- Contents of a dictionary are unordered.
- Keys are unique.

```
>>> colors = {'red': (1, 0, 0),
...           'green': (0, 1, 0)}
>>> colors['red']                  # lookup value
(1, 0, 0)
>>> colors['blue'] = (1, 0, 0)  # add new element
>>> colors
{'blue': (0, 0, 1), 'green': (0, 1, 0),
 'red': (1, 0, 0)}
```

Dictionaries

Two ways to check if the dictionary has a specific key:

```
>>> colors.has_key( 'red' )
True
>>> 'red' in colors
True
```

Delete an element from the dictionary:

```
>>> del colors['red']
>>> colors
{'blue': (0, 0, 1), 'green': (0, 1, 0)}
```

Dictionaries

To construct a dictionary from a list of tuples:

```
>>> dict([('seismology', 254),
...       ('seismics', 250),
...       ('vulcanology', 253)])
{'vulcanology': 253, 'seismics': 250,
 'seismology': 254}
```

A shortcut, when the keys are simple strings:

```
>>> config = dict( verbose=True,
                   recursive=False,
                   inputfile='data.txt' )
>>> config
{'inputfile': 'data.txt', 'verbose': True,
 'recursive': False}
```

Looping a dictionary

Efficient:

```
>>> knights = {'gallahad': 'the pure',
               'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
robin the brave
gallahad the pure
```

Sorted by keys:

```
keys = knights.keys()
keys.sort()
for k in keys:
    print k, knights[k]
```

Looping a dictionary

Sorted by values:

```
def by_value(a,b):
    return cmp(knights[a],knights[b])

keys = knights.keys()
keys.sort(by_value)
for k in keys:
    print k, knights[k]
```

Or short, using a *lambda form*:

```
keys = knights.keys()
keys.sort(lambda a,b :
    return cmp(knights[a],knights[b]))
for k in keys:
    print k, knights[k]
```

Things to remember from today's lesson

- Others have variables.
- Python has identifiers (which are references to objects).
- The stairways to power are lists and dictionaries.

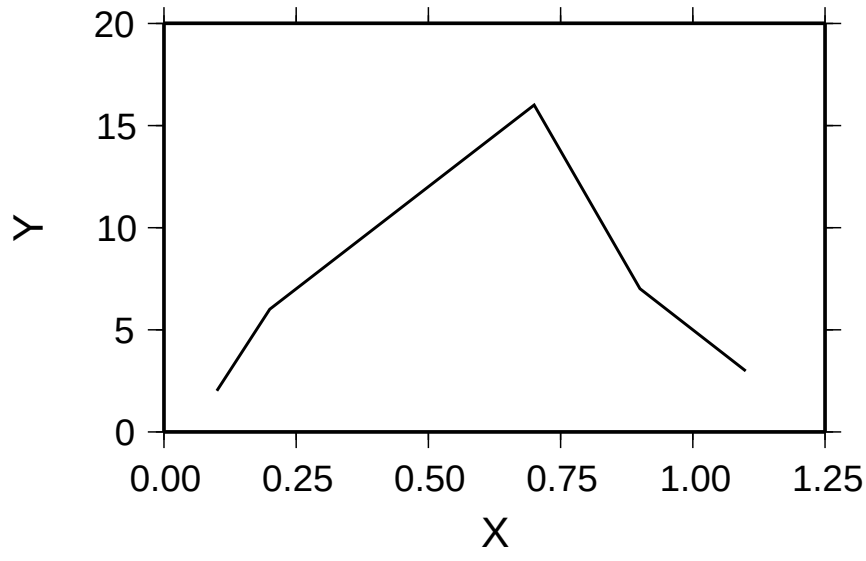
Keep these pages open, when programming in Python:

- Tutorial:
  `http://docs.python.org/tut/`
- Library Reference:
  `http://docs.python.org/lib/lib.html`

Training manoeuvre

Problem: plot autoscaling

X	Y
0.1	2
0.2	6
0.7	16
0.9	7
1.1	3



Picking a good tick increment for plotting is simple for us.
But how can this be automatized?

Training manoeuvre

Problem: plot autoscaling

- Design and implement a function which takes the minimum and maximum of some plot data, and returns a nice tick increment as well as a minimum and a maximum for a plot axis.
- Allow for some optional fine tuning like:
  - Specification of an approximate number of ticks.
  - Symmetry around zero.
  - Snap plot range to ticks.
  - Increase range by some small percentage.