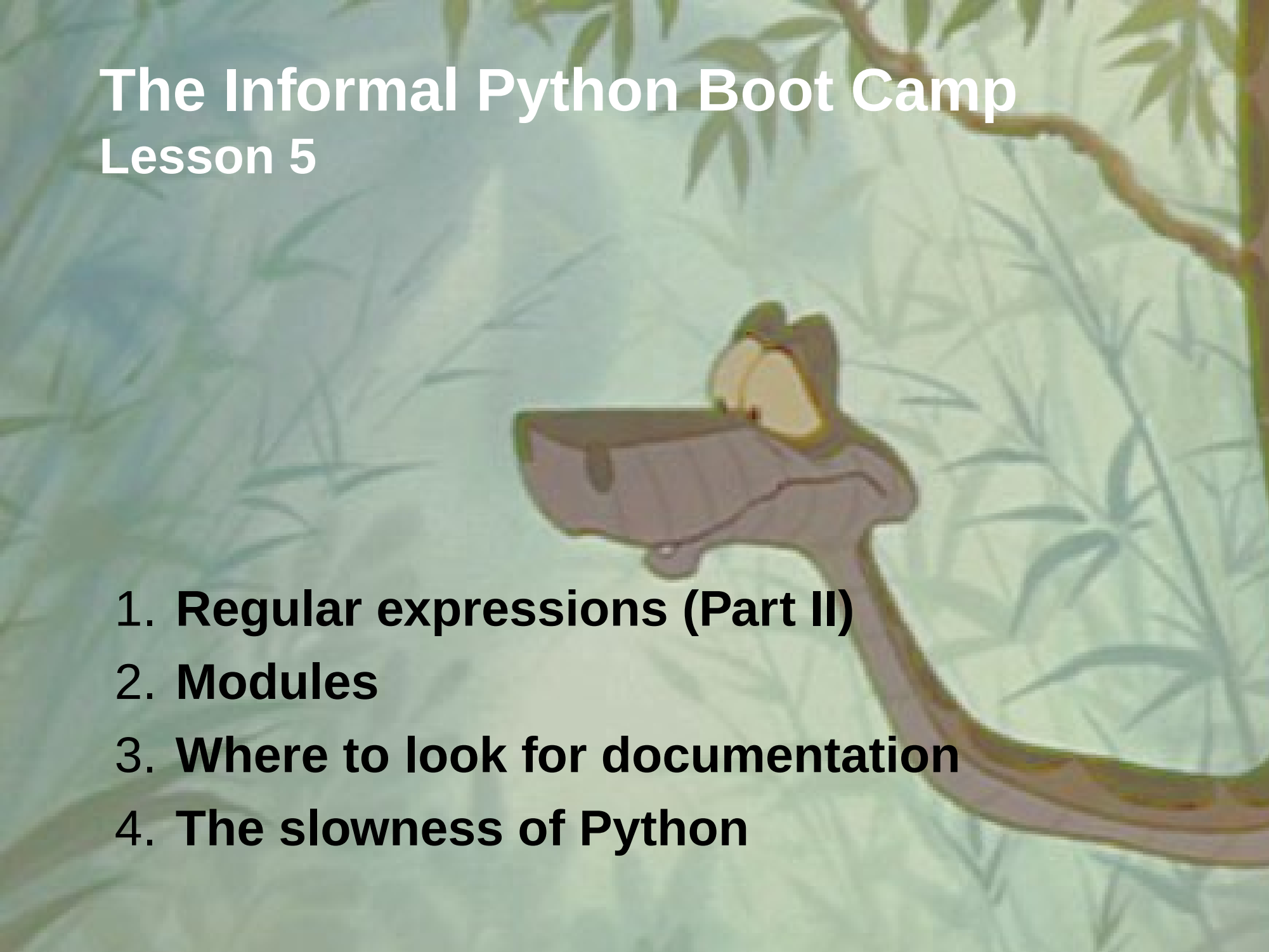

The Informal Python Boot Camp
Lesson 5

1. **Regular expressions (Part II)**
2. **Modules**
3. **Where to look for documentation**
4. **The slowness of Python**

Main functions provided by the re module

Overview

`re.findall(...)`	returns all matches as a list of strings.
`re.search(...)`	returns a *match object* for first match.
`re.finditer(...)`	returns *match objects* for each match.
`re.split(...)`	splits string where the pattern matches.
`re.sub(...)`	replaces matching substrings.

Also available as:

- methods of objects returned by `re.compile(...)`

```
>>> re.findall(r'\d+', "lat=12, lon=11")
['12', '11']

>>> compiled_pattern = re.compile(r'\d+')
>>> compiled_pattern.findall("lat=12, lon=11")
['12', '11']
```

Case insensitive search

- Most re functions take an additional flags argument.
- The most useful flag is re.I (= re.IGNORECASE).

```
>>> re.findall(r'so', 'Mal so mal So')
['so']
>>> re.findall(r'so', 'Mal so mal So', re.I)
['so', 'So']
```

Capturing parts of a match

Use parantheses to capture parts of a match:

```
>>> line = 'event: 2007/10/18  03:19:55  M=5.5'

>>> m = re.search('(\d+):(\d+):(\d+)', line)

>>> int(m.group(1))
3
>>> int(m.group(2))
19
>>> int(m.group(3))
55
```

Capturing parts of a match

Nested groups

```
>>> line = 'event: 2007/10/18  03:19:55  M=5.5'

>>> m = re.search('((\d+:\d+):(\d+))', line)
# group number:    (------- 1 -----)
#                   (---2---)
#                             (-3-)

>>> m.group(1)
'03:19:55'
>>> m.group(2)
'03:19'
>>> m.group(3)
'55'
```

Count opening parantheses for group number.

Replacing: sub(pattern, replacement, str)

Convert DOS and Mac line endings to UNIX line feeds:

```
>>> text = re.sub('\r\n?', '\n', alien_text)
```

Remove any filename suffix:

```
>>> re.sub('\.[^.]+$', '', 'lesson5.tex')
'lesson5'
```

Replace multiple whitespace with single spaces:

```
>>> re.sub('\s+', ' ', """Hooray
                           for \t  Dvorak!""")
'Hooray for Dvorak!'
```

Matching valid floating point numbers

```
22
5.202
+1.302
-5.2
.123
2e-3
2.E-3
```

Use this as a recipe:

```
[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?
```

```
METACHARS                    FLAGS
^     string begin           re.I case insens.
$     str. end (before \n)   re.M line based ^$
+     one or more            re.S . includes \n
*     zero or more           re.X ign. wh.space
?     zero or one            re.L locale aware \w,\b,\s
{3,7} repeat in range
()    capture              CHARCLASSES
(?:)  no capture           .  == [^\n]
[]    character class      \s == [ \t\n\r\f\v]
|     alternation          \w == [A-Za-z0-9_]
\b    word boundary        \d == [0-9]
\Z    string end           \S, \W and \D negate

FLOATS: [-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?

FUNCTIONS  p=pattern, s=string, f=flags, c=count
re.findall(p,s[,f])   returns list of strings
re.search(p,s[,f])    returns first match as match obj.
re.finditer(p,s,[,f]) iterates over match objects
re.split(p,s,[,c])    returns list of strings
re.sub(p,r,s[,c])     replaces matching substrings
re.escape(s)          escape regex metacharacters
```

Importing functionality of a module

The normal and safe aproach:

```
>>> import math
>>> math.pi
3.1415926535897931
>>> math.cos(math.pi)
-1.0
```

Not for the faint-hearted:

Importing directly into the local namespace:

```
>>> from math import *
>>> pi
3.1415926535897931
>>> cos(pi)
-1.0
```

Caution with `from module import *`

Consider different modules providing functions with the same name.

Example name collision:

```
>>> from os import *
# open() is now mapped to os.open() !!!

>>> f = open( "lesson5.tex", "r" )  # fails!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: an integer is required
```

The builtin open() could still be accessed though:

```
>>> f = __builtins__.open("lesson5.tex","r")
```

Less typing while avoiding collisions:

Binding to local names:

```
>>> import math
>>> cos = math.cos
>>> pi = math.pi
>>> cos(pi)
-1.0
```

Import module under a different/shorter name:

```
>>> import math as m
>>> m.cos(m.pi)
-1.0
```

Import only what is needed:

```
>>> from math import pi, cos
>>> cos(pi)
-1.0
```

Writing your own module

seismo.py:

```
"""Some seismological utility functions."""

import math

def lame_parameters( alpha, beta, density ):
    """Convert seismic velocities to Lame's parameters.

       Returns Lame's parameters as (lambda, mu)."""

    return ( (alpha**2-2.0*beta**2)*density,
             beta**2*density )

def velocities( lambd, mu, density ):
    """Convert lame parameters to seismic velocities.

       Returns tuple with velocities (alpha, beta)."""

    return ( math.sqrt((lambd+2.0*mu)/density),
             math.sqrt(mu/density) )
```

Using your module

As with any other module:

```
>>> import seismo

>>> seismo.lame_parameters( 4000., 2100., 2600. )
(18668000000.0, 11466000000.0)

>>> seismo.velocities( *(_+(2600,)) )
(4000.0, 2100.0)
```

Help!

```
>>> import seismo
>>> help(seismo)  # makes man page from docstrings!
```

```
    Help on module seismo:

NAME
    seismo - Some seismological utility functions.

FILE
    /scratch/local1/sebastian/home-ext/bootcamp/seismo.py

FUNCTIONS
    lame_parameters(alpha, beta, density)
        Convert seismic velocities to Lame’s parameters.

        Returns Lame’s parameters as (lambda, mu).

    velocities(lambd, mu, density)
        Convert lame parameters to seismic velocities.

        Returns tuple with velocities (alpha, beta).
```

Introspection

You can look at the contents of any module:

```
>>> import seismo
>>> dir(seismo)
['__builtins__', '__doc__', '__file__', '__name__',
 'lame_parameters', 'math', 'velocities']

 # dir without argument looks at local namespace:
 >>> dir()
 ['__builtins__', '__doc__', '__name__', 'seismo']
```

Eight most essential modules

sys	`argv`, `stdin`, `stdout`, `stderr`, `exit()`
os	Directory, file and process management
shutil	Higher level copying/moving of files
math	Standard mathematical functions
random	Several random number generators
time	Time and Date manipulation, `sleep()`
re	Regular expressions
subprocess	Process management, IPC with pipes

Where to search for documentation

- `pydoc modulename`
- Python library reference
- Python library reference index
- Python global module index

Python is extremely slow and wastes memory!

```
import os
xvec = range(2000000)
yvec = range(2000000)
zvec = [ 0.5*(x+y) for x,y in zip(xvec,yvec) ]

os.system( "ps v "+str(os.getpid()) )
```

- 104 MB memory usage!
- Runs almost 8 seconds!

Comparison with C

```c
#include <stdlib.h>
#include <unistd.h>
main () {
    int *avec, *bvec;
    float *cvec;
    int i, n = 2000000;
    avec = (int*)calloc( n, sizeof(int) );
    bvec = (int*)calloc( n, sizeof(int) );
    for (i=0;i<n;i++) {
        avec[i] = bvec[i] = i;
    }
    cvec = (float*)calloc( n, sizeof(float) );
    for (i=0;i<n;i++) {
        cvec[i] = (avec[i]+bvec[i])*0.5;
    }
    int slen = 100;
    char *command = (char*)calloc( slen, sizeof(char) );
    snprintf(command, slen,  "ps v %i", getpid() );
    system( command );
}
```

- 25 MB memory usage.
- Runs about 0.1 s (almost a hundred times faster!)

When speed matters

- Use specialized array math modules: numpy, scipy.
- Write or use C/Fortran extensions for time critical tasks.

We will see more on these topics in later sessions.

Numpy example

```
import os
from numpy import *

xvec = arange( 2000000 )
yvec = arange( 2000000 )
zvec = (xvec+yvec)*0.5

os.system( "ps v "+str(os.getpid()) )
```

- 37 MB memory usage
- Runs about 0.3 s

Things to remember from today's lesson

- `help()`, `dir()` and `pydoc module`
- Python itself is slow.
- Python regex HOWTO:
  `http://www.amk.ca/python/howto/regex/`

Keep these pages open, when programming in Python:

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