The Informal Python Boot Camp
Lesson 2

1. **Python unleashed**
2. **Indentation**
3. **Flow control**
4. **Tell me your name and I tell you who you are.**

Python unleashed

my_first_test.py:

```
good_foods = [ 'Schnitzel', 'Pommes' ]
bad_foods = [ 'Salat' ]

menu = ['Schnitzel', 'Pommes', 'Eis', 'Salat']
for food in menu:
    if food in good_foods:
        print "good:", food

    if food in bad_foods:
        print "oh no!", food
```

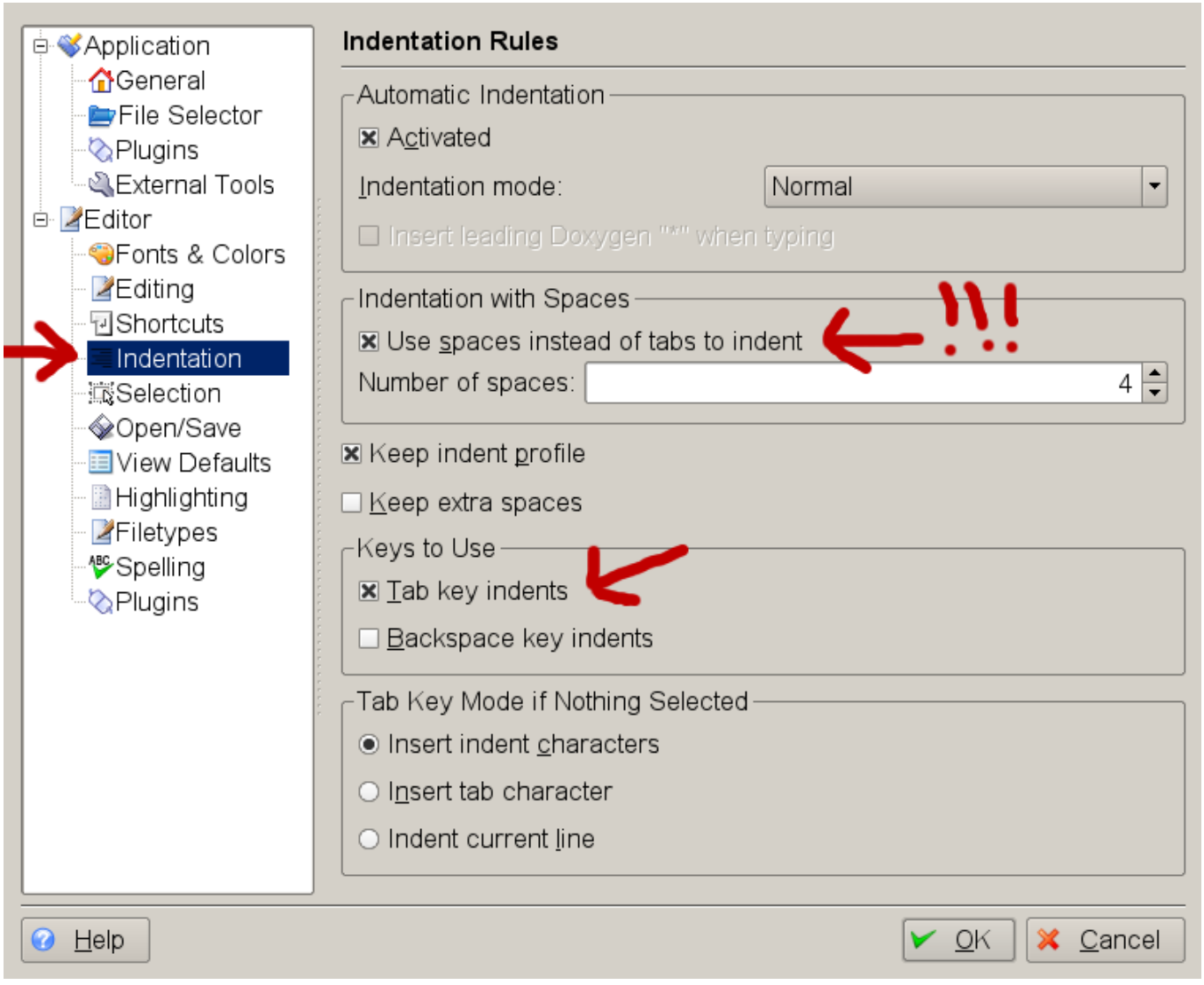
```
% python my_first_test.py
good: Schnitzel
good: Pommes
oh no! Salat
```

Indentation

Not using indentation is a capital sin when programming!

- Anyway, indentation is obligatory in Python.
- Use spaces. Never use tabs!
- Tab behaviour is not deterministic!
- I recommend an indentation level of 4 spaces.

Tabs are evil! Make your editor behave!



Convert tabs to spaces on existing code.

```
% sed 's/\t/    /g' <bad.py >good.py
```

Flow control

We will now again loosely follow the official Python Tutorial: Chapter 4, *More Control Flow Tools*:

- `http://docs.python.org/tut/`

If statements

```
x = int(raw_input("Please enter an integer: "))
if x < 0:
     x = 0
     print ’Negative changed to zero’
elif x == 0:
     print ’Zero’
elif x == 1:
     print ’Single’
else:
     print ’More’
```

What is true - what is false

- Non-zero integers are *true*.
- Zero is *false*.
- Any sequence with a non-zero length is *true*.
- Empty sequences are *false*.

```
import sys
files  = sys.argv[1:]
if not files:
    # missing command line arguments
    print "usage: dosomething.py filename ..."
    sys.exit(1)

# do something with files...
```

While statement

Let's see how far we can count in Python:

```
>>> import time
>>> i = 1
>>> while True:
...     i = i * 1000   # same as: i *= 1000
...     print repr(i)
...     time.sleep(1)  # wait one second
...
1000
1000000
1000000000
1000000000000L   # <- type conversion occured!
1000000000000000L
# ... continues until memory is exhausted!
```

For statements

The `for` statement iterates over the items of any sequence:

```
menu = ['Schnitzel', 'Pommes', 'Eis', 'Salat']
for food in menu:
    print food

for letter in 'habakuk'
    print letter
```

For statements

But don’t change the list you are iterating over!

Use a copy of the list in case:

```
>>> menu = ['Schnitzel', 'Pommes']
>>> for food in menu[:]:
...     menu.append( food )
>>> menu
['Schnitzel', 'Pommes', 'Schnitzel', 'Pommes']
```

The range function

Range creates a list of numbers:

```
>>> for i in range(4):
...     print i, 2**i
...
0 1
1 2
2 4
3 8
```

The range function behaves similar to slicing operators:

```
>>> range(4)
[0, 1, 2, 3]
>>> range(3,7)
[3, 4, 5, 6]
>>> range(2,10,2)
[2, 4, 6, 8]
```

The continue statement

The `continue` statement brings you to the next iteration:

```
import sys
files  = sys.argv[1:]
for file in files:
    if file[-4:] != '.jpg':   # skip non-jpeg files
        continue

    # do something with the jpeg file
    blablabla(file)
```

The break statement

The `break` statement let's you abort a loop abnormally:

```
good_foods = [ 'Schnitzel', 'Pommes' ]
menu = ['Schnitzel', 'Pommes', 'Eis', 'Salat']
wanted = []
for food in menu:
    if food in good_foods:
        wanted.append( food )

    if len(wanted) == 2:
        break  # two things are enough today...

print "ich will: " + ' '.join(wanted)
```

The pass statement

The `pass` statement does nothing:

```
>>> while True:
...     pass # Busy-wait for keyboard interrupt
...
```

It can be used when a statement is required syntactically but the program requires no action.

Defining functions

The `def` keyword introduces a function:

```
>>> def powers_of_two(n):
...     results = []
...     for i in range(n):
...         results.append( 2**i )
...     return results
...
>>> print powers_of_two(10)
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

Default argument values and keyword arguments

A function may be called with a variable number of arguments:

```
def ask_ok( prompt, retries=4,
            complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'): return True
        if ok in ('n', 'no'): return False
        retries = retries - 1
        if retries < 0:
            raise IOError, 'refusing user'
        print complaint

ask_ok('Do you really want to quit?')
ask_ok('OK to overwrite the file?', 2)
ask_ok('Coffee?', complaint='Yes or no, dude!')
```

What might this piece of code do?

```
a = gfns()
b = []
for i in a:
    p = rplxt(i,xt)
    b.append( p )
```

And what does this code do?

```
input_files = get_filenames()
output_files = []
for ifile in input_files:
    ofile = replace_extension( ifile, extension )
    output_files.append( ofile )
```

The other extreme...

```
list_of_input_files = get_list_of_filenames()
list_of_output_files = []
for input_file in list_of_input_files:
    output_file = replace_extension( input_file,
                                     filename_extension )
    list_of_output_files.append( output_file )
```

Use good names!

This is trick number one to write readable code.

Naming is not easy. Think before you type.

Naming identifiers

Names should within their context be self-explaining

- Use complete names for variables in a large scope.

```
earth_radius = 6.371e6
```

- Variable names in a local scope can often be made short without loss of readability:

```
for ifile in input_files:
    print ifile
```

Naming identifiers

Use taste

- Never use cryptic abbreviations.
- Names should not be too long - make a compromise.
- If in doubt: use long names.
- Use local alias variables when having trouble with too long names:

```
l = lame_lambda
m = lame_mu
d = density
youngs_modulus  = (3.0*l + 2.0*m)*m / (l+m)
p_wave_velocity =  math.sqrt((l+2.0*m) / d)
```

Naming identifiers

Names should within their context be self-explaining

This applies to more than just variable names:

- Function names
- Module names
- File names - DOS 8.3 is history!
- Email subjects

Naminging conventions

Naming conventions are subject to personal taste if not enforced by project guidelines.

General goals:

- Consistency
- Readability

Adapt your own style!

Example naming conventions

- Use underscores:

```
input_files = get_filenames()
output_files = []
for ifile in input_files:
    ofile = replace_extension( ifile, extension )
    output_files.append( ofile )
```

Example naming conventions

- Camel-case:

```
inputFiles = getFilenames()
outputFiles = []
for iFile in inputFiles:
    oFile = replaceExtension( iFile, extension )
    outputFiles.append( oFile )
```

Example naming conventions

Use different decorations to mark special sets of identifiers. For example:

- All-uppercase letters for constants
- Capitalized names of classes
- Leading underscore on global variables

Example naming conventions

I think this is extremely useful for numerical programming:

- Number of items begin with letter `n`
- Iterators begin with letter `i`

Example:

```
for irec in range(nreceivers):
    for icomp in range(ncomponents):
        seismogram[irec][icomp] = ...
```

This has reduced debugging work by about 60% for me!

Real world FORTRAN.

If you can find the mistake ”hands down”, you may have Dr habil. skills ;-)

```
DO WHILE (ALIVE.LT.N*M)
CALL POPHEAP(HEAP,MIN,TIMES,BP)
IF(MIN.EQ.0) GOTO 666
I=MOD(MIN-1,N)+1
J=(MIN-1)/N+1
BP(I ,J)=ISALIVE
ALIVE=ALIVE+1
IF(1 .LT. I)CALL UNBR(I-1,J)
IF( N.GT.I)CALL UNBR(I+1,J)
IF(1.LT.J) CALL UNBR(I,J-1)
IF(N.GT.J)CALL UNBR(I,J+1)
END DO
```

Improved version: But the mistake is still difficult to find...

```
do while (alive < n*m)

  ! get the narrowband element with the
  ! smallest value for T and make it alive
    call popheap( heap, min, times, backpointers )
    if (min == 0) exit
    i = mod(min-1,n) + 1
    j = (min-1)/n + 1
    backpointers(i,j) = IS_ALIVE
    alive = alive + 1

  ! update adjacent nodes
    if (1 < i)  call update_neighbor(i-1,j)
    if (i < n)  call update_neighbor(i+1,j)
    if (1 < j)  call update_neighbor(i,j-1)
    if (j < n)  call update_neighbor(i,j+1)
end do
```

Employing the i/n-convention makes things clearer...

```
do while (nalive < nx*ny)

  ! get the narrowband element with the
  ! smallest value for T and make it alive
    call popheap( heap, imin, times, backpointers )
    if (imin == 0) exit
    ix = mod(imin-1,nx) + 1
    iy = (imin-1)/nx + 1
    backpointers(ix,iy) = IS_ALIVE
    nalive = nalive + 1

  ! update adjacent nodes
    if ( 1 < ix) call update_neighbor( ix-1, iy   )
    if (ix < nx) call update_neighbor( ix+1, iy   )
    if ( 1 < iy) call update_neighbor( ix,   iy-1 )
    if (iy < nx) call update_neighbor( ix,   iy+1 )
end do
```

Correct and readable (though still not understandable) version

```
do while (nalive < nx*ny)

  ! get the narrowband element with the
  ! smallest value for T and make it alive
    call popheap( heap, imin, times, backpointers )
    if (imin == 0) exit
    ix = mod(imin-1,nx) + 1
    iy = (imin-1)/nx + 1
    backpointers(ix,iy) = IS_ALIVE
    nalive = nalive + 1

  ! update adjacent nodes
    if ( 1 < ix) call update_neighbor( ix-1, iy   )
    if (ix < nx) call update_neighbor( ix+1, iy   )
    if ( 1 < iy) call update_neighbor( ix,   iy-1 )
    if (iy < ny) call update_neighbor( ix,   iy+1 )
end do
```

Naming conventions in Python

There is an ”official” style guide for Python code:

- `http://www.python.org/dev/peps/pep-0008/`

More on programming style in Python:

- `http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html`

Things to remember from today's lesson

- Don't use tabs.
- Names should within their context be self-explaining.

Keep these pages open, when programming in Python:

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