Introduction To Python Programming

I: The Python language

Jan Moren, SCDA

OIST

OKINAWA INSTITUTE OF SCIENCE AND TECHNOLOGY GRADUATE UNIVERSITY
沖縄科学技術大学院大学

Python is High Level, Interpreted and Dynamic

- No memory management
- No system programming needed
- Complex data structures
- Extensive library of modules ("batteries included")

- Standalone programs
- Scripting language
- Interactive use
- Embeddable into applications
- OS and system agnostic

Useful *everywhere*

Written in or Extends Python:

PyMOL	TensorFlow
NEST	NEURON
Blender	Qiime
HTSeq	BUSCO
...	...

Run Python on:

Windows	Supercomputers
Linux	Desktops
OSX	Microcontrollers
Android	

Focus on your problem, not your code

Example: Replace text strings in a list of files

- Python version below

Python

```
import fileinput

files=["test1.txt", "test2.txt", "test3.txt"]

for line in fileinput.input(files, inplace=True):
    print(line.replace('Goodbye!', 'Hello!'), end='')
```

Focus on your problem, not your code

Example: Replace text strings in a list of files

- Python version below
- C to the right

Python

```python
import fileinput

files=["test1.txt", "test2.txt", "test3.txt"]

for line in fileinput.input(files, inplace=True):
    print(line.replace('Goodbye!', 'Hello!'), end='')
```

C

```c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <string.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <err.h>
#include <string.h>

char * find_match(const char *buf, const char * buf_end, const char *pat, size_t len)
{
        ptrdiff_t i;
        char *start = buf;
        while (start + len < buf_end) {
                for (i = 0; i < len; i++)
                        if (start[i] != pat[i]) break;

                if (i == len) return (char *)start;
                start++;
        }
        return 0;
}

int replace(const char *from, const char *to, const char *fname)
{
#define bail(msg) { warn(msg" '%s'", fname); goto done; }
        struct stat st;
        int ret = 0;
        char *buf = 0, *start, *end;
        size_t len = strlen(from), nlen = strlen(to);
        int fd = open(fname, O_RDWR);

        if (fd == -1) bail("Can't open");
        if (fstat(fd, &st) == -1) bail("Can't stat");
        if (!(buf = malloc(st.st_size))) bail("Can't alloc");
        if (read(fd, buf, st.st_size) != st.st_size) bail("Bad read");

        start = buf;
        end = find_match(start, buf + st.st_size, from, len);
        if (!end) goto done; /* no match found, don't change file */

        ftruncate(fd, 0);
        lseek(fd, 0, 0);
        do {
                write(fd, start, end - start);      /* write content before match */
                write(fd, to, nlen);                /* write replacement of match */
                start = end + len;                  /* skip to end of match */
                                                    /* find match again */
                end = find_match(start, buf + st.st_size, from, len);
        } while (end);

        /* write leftover after last match */
        if (start < buf + st.st_size)
                write(fd, start, buf + st.st_size - start);

done:
        if (fd != -1) close(fd);
        if (buf) free(buf);
        return ret;
}

int main()
{
        const char * files[] = { "test1.txt", "test2.txt", "test3.txt" };

        for (int i = 0; i < sizeof(files)/sizeof(char*); i++)
                replace("Goodbye!", "Hello!", files[i]);

        return 0;
}
```

OIST

Let’s Run some Python

We will use “ipython” for an interactive shell

- Log in to Deigo
- load Python 3
- start iPython

```
ssh -X <your-name>@deigo.oist.jp

module load python/3.7.3

ipython3
```

History Of Python
— or —
Why are there two of them?!

```
ssh -X <your-name>@deigo.oist.jp
module load python/3.7.3
ipython3
```

Python 2

- Named after Monty Python
- released in 2000
- Huge success; many users, lots of packages

- **End of life 2020**

  → never use it any longer

Python 3

- first released in 2008
- 3.5 onwards is now stable, complete, supported

- **We Recommend Python 3 for all future Python projects.**

Example

Code comes in *blocks*

- marked by indentation, not "{","}" or "begin", "end"

if, while, function defs etc. end with ":", followed by a block

you don't need to declare variable types

starts a comment

```
# an example python program

def greeting(name, age):
    print("Hi {}.".format(name))

    if age>45:
        print("You must be very wise!")

# here the main program starts
while True:
    name = input('your name (or quit) :')
    if name=='quit':
        break

    age = input("what's your age :")
    greeting(name, int(age))

print('Bye now!')
```

Example

Code comes in *blocks*

- marked by indentation, not "{","}" or "begin", "end"

if, while, function defs etc. end with ":", followed by a block

you don't need to declare variable types

starts a comment

```
# an example python program

def greeting(name, age):
    print("Hi {}.".format(name))

    if age>45:
        print("You must be very wise!")

# here the main program starts
while True:
    name = input('your name (or quit) :')
    if name=='quit':
        break

    age = input("what's your age :")
    greeting(name, int(age))

print('Bye now!')
```

Example

Code comes in *blocks*

- marked by indentation, not "{","}" or "begin", "end"

if, while, function defs etc. end with ":", followed by a block

you don't need to **declare variable types**

starts a comment

```
# an example python program

def greeting(name, age):
    print("Hi {}.".format(name))

    if age>45:
        print("You must be very wise!")

# here the main program starts
while True:
    name = input('your name (or quit) :')
    if name=='quit':
        break

    age = input("what's your age :")
    greeting(name, int(age))

print('Bye now!')
```

Example

Code comes in *blocks*

- marked by indentation, not "{","}" or "begin", "end"

if, while, function defs etc. end with ":", followed by a block

you don't need to declare variable types

 # starts a comment

```
# an example python program

def greeting(name, age):
    print("Hi {}.".format(name))

    if age>45:
        print("You must be very wise!")

# here the main program starts
while True:
    name = input('your name (or quit) :')
    if name=='quit':
        break

    age = input("what's your age :")
    greeting(name, int(age))

print('Bye now!')
```

Numbers

Integers

- arbitrarily large

Floating point numbers

- 64 bit

- Complex numbers
  - pair of floats
  - use ‘j’ for imaginary part

```
  1+1             # integers
2
  1.5*2.5         # floating point
3.75
  5/3             # division
1.6666666666666667
  5//3            # integer division
1
  5%3             # remainder
2
  2.5**4          # exponentiation
39.0625
  4+3j            # complex numbers
(4+3j)
  int(3.5)        # convert types
3
  float(4)
4.0
```

Variables

Variables

- stores a value
- don't need to declare the type
  - can force a type with type functions:

```
  x = 1                    # x is an int
  type(x)                  # x is an int
int
  y = float(x)             # y is a float
  type(y)
float
  y
1.0
```

```
  x = 1                    # x is now an int
  y = 1.5                  # y is a float
  z = 3+1j                 # z is a complex
  res = x+y*z              # res is complex

  print("res:", res) # print the value
res: (5.5+1.5j)

  x = x+2                  # increase x
  x += 2                   # same thing
```

Variables

Variables

- stores a value
- don't need to declare the type
  - can force a type with type functions
- most things ("objects") in Python have *attributes* and *methods*
  - **attribute**: an associated value
  - **method**: object-specific function

```
  x = 1                  # x is now an int
  y = 1.5                # y is a float
  z = 3+1j               # z is a complex
  res = x+y*z            # res is complex

  print("res:", res) # print the value
res: (5.5+1.5j)

  x = x+2                # increase x
  x += 2                 # same thing

  res.real               # an attribute
5.5
  res.conjugate()    # a method
(5.5-1.5j)
```

Booleans and comparisons

comparison operators

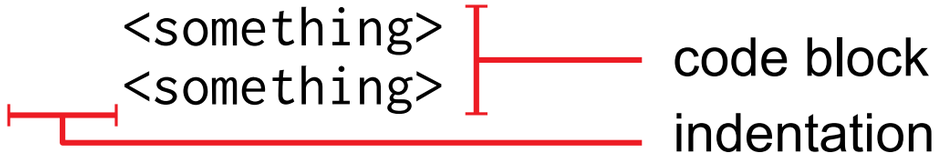
- all comparisons return True or False.
- "bool" is the type for True and False
- (True is 1, False is 0)
- "and", "or" and "not" combine boolean values:
  - a and b : true if a and b are true
  - a or b : true if either a or b is true
  - not a : true if a is false, false if a is true

```
  x = 5
  y = 3

  x > y          # greater than?
True
  x == y         # Equal? Note: two '=='
False
  x != y         # Not equal?
True
  x <= y         # smaller or equal?
False
  x==y or y>0    # either or
True
  x>0 and x<10 # both
True

  x = y<0        # can store booleans
  print(x)
False
```

the "if" statement

Do something only "if" true:

```
if <boolean> :
    <something>
    <something>
```

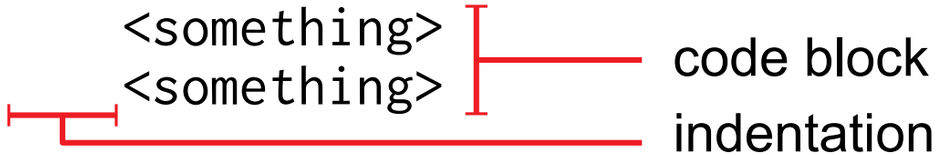


- a code *block* is a group of code lines with the same indentation
  - be careful that you use the exact same number of spaces! Usually 4 per level.
- if the boolean condition is true: run the code in the block

the "if" statement

Do something only "if" true:

```
if <boolean> :
    <something>
    <something>

else:
    <something else>
    <something else>
```

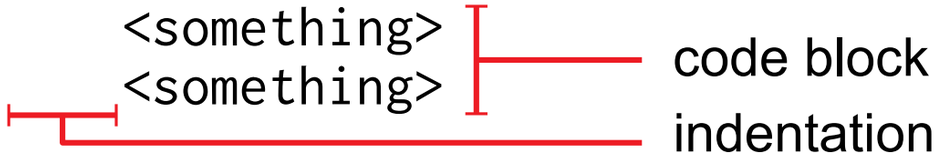


- a code *block* is a group of code lines with the same indentation
  - be careful that you use the exact same number of spaces! Usually 4 per level.
- if the boolean condition is true: run the code in the block
- "else:" optional
  - runs the "else" block if condition failed

the "if" statement

Do something only "if" true:

```
if <boolean> :
    <something>
    <something>

elif <boolean> :
    <something>
    <something>

else:
    <something else>
    <something else>
```



- a code *block* is a group of code lines with the same indentation
  - be careful that you use the exact same number of spaces! Usually 4 per level.
- if the boolean condition is true: run the code in the block
- "else:" optional
  - runs the "else" block if all conditions failed
- "elif:" optional, and short for "else if".
  - only tested if the previous test fails
  - can have as many as you want

the "if" statement

Do something only "if" true:

```
x = -3

if x>0:
    print("positive")

elif x<0:
    print("negative")

else:
    print("zero")
```

- a code *block* is a group of code lines with the same indentation
  - be careful that you use the exact same number of spaces! Usually 4 per level.
- if the boolean condition is true: run the code in the block
- "else:" optional
  - runs the "else" block if all conditions failed
- "elif:" optional, and short for "else if".
  - only tested if the previous test fails
  - can have as many as you want

the "if" statement

Exercise: How do you test if a value is odd?

- Hint: '%' the remainder operator
  - gives the remainder of a division

the "if" statement

Exercise: How do you test if a value is odd?

```
if x%2 == 1:
    print("odd")
```

- Hint: '%' the remainder operator
  - the remainder of an integer division by 2 is 1 if the numerator is odd

the "if" statement

Exercise: How do you test if a value is odd?

```
if x%2 == 1:
    print("odd")
```

— or —

```
if x%2:
    print("odd")
```

- Hint: '%' the remainder operator
  - the remainder of an integer division by 2 is 1 if the numerator is odd
  - '1' is True, so we don't need the test to see if it's 1.

    But it is often better to be explicit.

looping: the "while" statement

```
while <condition>:
    <code block>
    <code block>
```

- Repeat a code block "while" condition is true

`break`	jump out of block
`continue`	go back to top of block

```
i = 0
while i<10:
    print("val:", i)
    i += 1
```

- Example: loop from 0 to 9 and print each value

Work with Python

How do we actually do things?

- **iPython** good for direct experimentation or as a calculator
- **iPython+ code file** Interactive development – run, debug and so on
- **IDE** Whole development environment
- **Jupyter** Interactive notebooks

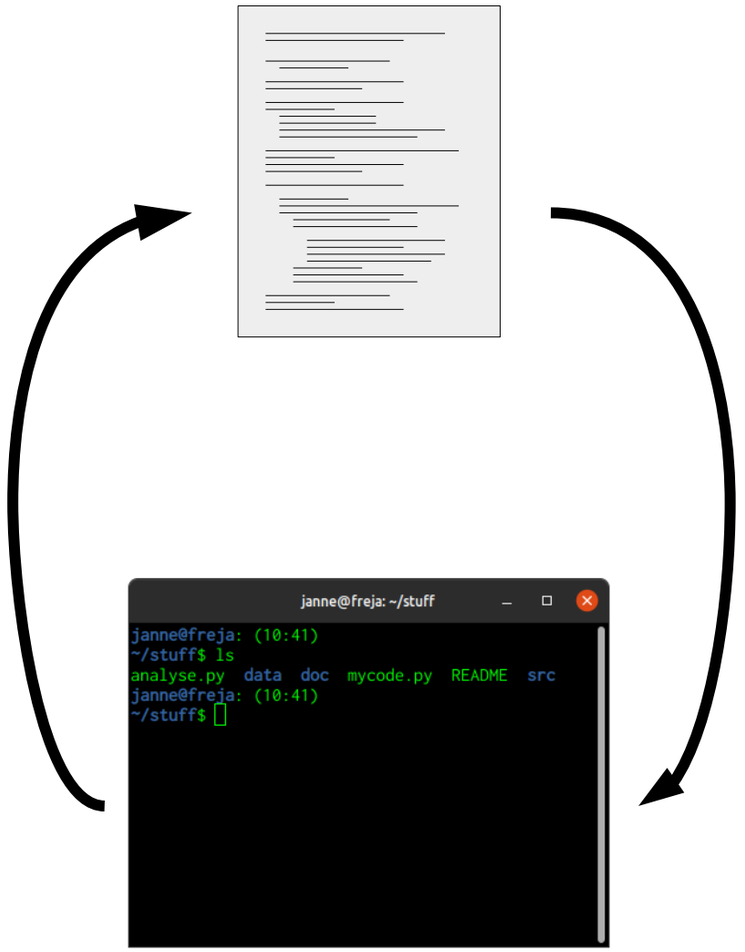
Put your code in a file

Have your code in a file

- Much easier to edit as the code becomes larger
- Can use the code as a regular program

The edit - (compile) - run cycle

1. edit your code in editor
2. run in iPython (or directly in terminal)
3. check results, play around
4. go back to 1.



Put your code in a file

1. Open another terminal
   - log in, load python module
   - Run either:

```
nano oddloop.py
```

   or

```
gedit oddloop.py
```

   - Edit your file

gedit:

- Go to preferences->editor, then set:
  - ✔ tab width 4
  - ✔ insert spaces instead of tabs
  - ✔ enable automatic indentation

In general, you want tab width 4 and spaces for tabs in any editor

Put your code in a file

1. Open another terminal
   - log in, load python module
   - Run either:

     ```
     nano oddloop.py
     ```

     or

     ```
     gedit oddloop.py
     ```

   - Edit your file

```
i = 0
while i<10:
    print("val:", i)
    i += 1
```

Put your code in a file

1. Open another terminal
   - log in, load python module
   - Run either:

```
nano oddloop.py
```

or

```
gedit oddloop.py
```

```
i = 0
while i<10:
    print("val:", i)
    i += 1
```

Run the code:

- Open *another* terminal

log in, load python module

- Run:

```
python3 oddloop.py
```

- Or, in iPython:

```
%run oddloop.py
```

iPython tips:

you can edit multi-line "cells"

- automatically indents blocks
- press enter on last line to run.

```
In [4]: while i<10:
   ...:     print("val:", i)
   ...:     i += 1
```

Advanced:

- `%edit` edits cells in vim editor
- `%debug` runs code in debugger

`%run` will run a python program

- Will stop at the end, let you examine variables, etc

```
%run oddloop.py
```

`%load` will load a file as if you had typed it in

- Can go back and edit, etc.

```
%load oddloop.py
```

Quick Exercise:

Print out the numbers from 0-9, saying if they're odd or even:

```
even: 0
odd:  1
even: 2
odd:  3
even: 4
odd:  5
even: 6
odd:  7
even: 8
odd:  9
```

- Use "while" and "if"
- remember “%” to test if odd or even

Quick Exercise:

Print out the numbers from 0-9, saying if they're odd or even:

```
even: 0
odd:  1
even: 2
odd:  3
even: 4
odd:  5
even: 6
odd:  7
even: 8
odd:  9
```

- Use "while" and "if"
- remember “%” to test if odd or even

```
i = 0

while i<10:
    if i%2 == 1:
        print("odd: ", i)
    else:
        print("even:", i)

    i += 1
```

Strings

- quote strings with " or '
  - ''' ... ''' are multi-line strings
- use \ to escape characters
  - **\' and \"** == " and '
  - **\n** == new line
  - **\t** == tab
  - **** == the '\' character itself

```
  s = "hi world!"    # a string
  s = 'hi world!'    # " or ' both work
  print(s)
hi world!

  s = 'I\'m\na string'  # '\' escapes
  print(s)              # characters
I'm
a string

  # multi-line strings
  s = '''first line
second line
'''
  print(s)
first line
second line
```

String interpolation

```
  greeting = 'Hello {}!'.format('World'))
  print(greeting)
Hello World!
```

You can "fill in" values into a string

- mark places in the string with '`{}`'
- use the '`format()`' method to set the value.

String interpolation

```
  'one: {} two: {}'.format(1,2)
one: 1 two: 2


  'two: {1} onetwo: {0} {1}'.format(1,2)
two: 2 onetwo: 1 2


  'int: {0:5d} int: {0:05d}'.format(123)
int:   123 int: 00123


  'f: {0:8.2f} {0:8.3e}'.format(1234.567)
float:  1234.57 1.23e+03
```

You can "fill in" values into a string

- mark places in the string with '`{}`'
- use the '`format()`' method to set the value.
- `{0}` = use first argument to `format()`
- `{:something}` - format specifier
  - `num` - field width
  - `.num` - decimals
  - `g,d,f,e,s...` - data type

String-related functions and methods

- use `input()` to ask for user input
- Many functions work on strings:
  - `len(s)` length of string
  - `int(s)` convert string to int
  - `float(s)` convert to float
- Strings have a lot of methods:
  - convert text
  - test what kind of text it is
  - find substrings
  - split strings

```
  s = input("name: ") # get user input
  print("Hi", s)
Hi Janne

  len(s)                # string length
5

  s.isalpha()           # lots of methods
True

  s.find('nn')          # find substring
2

  'nn' in s             # test substring
True                    # inclusion

  int("42")             # convert to other
42                      # types
```

Substrings and indexing

```
  s = "hi world!"

  print(s[3])           # get character
'w'                     # at pos 3
  print(s[3:6])         # from pos 3 to
'wor'                   # before 6
  print(s[-2])          # index from end
'd'

 s[3] = 'W'             # Error!

  s=s[:3]+'W'+s[4:]     # create new string
                        # instead
```

- get substrings with "`[...]`"
  - index starts at `0`
  - `[-x]` index from end:
    `[-1]` == last character
  - `[x:y]` == "from `x` to `y-1`"
  - `[:y]` == `[0:y]`
  - `[x:]` == `"from x to end"`
  - `[:]` == the entire string
- can't change strings ("immutable")
  - create copy instead

Lists

- a list of elements
  - elements can be any type, including lists.
  - you can mix different types in a single list
- `append()`, `pop()` adds and removes elements to the end. Fast.
  - `insert(i)` and `pop(index)` adds and removes from middle or start
    - ***much* slower than append**

```
  e = []                  # an empty list
  a = [1, 2, 3]           # a list of ints
  b = ['text', [7, 1.3]] # mixed list

  a.append(7)             # add an element
[1, 2, 3, 7]              # to the end

  elem = a.pop()          # remove and return
  print(elem, a)          # last element
7 [1, 2, 3]

  a.insert(1, 99)         # add or remove from
[1, 99, 2, 3]             # middle of list

   a.pop(1)
[1, 2, 3]
```

Lists

- Index list like with string
  - `[n]`, `[m:n]`, `[n:]` ...
-

```
  a = [1,2,3,4,5]
  a[0]                    # First element
1
  a[1:3]                  # a[1] and a[2]
[2, 3]

  a[2:]                   # to end
[3, 4, 5]

  a[:3]                   # from beginning
[1, 2, 3]

  a[-1]                   # count from back
5
```

Lists

- Index list like with string
  - `[n]`, `[m:n]`, `[n:]` ...
- Lists are mutable
- Many methods and functions:
  - `max()`, `min()`, `len()`, `sum()` ...
  - `l.sort()`, `l.index()`, `l.count()`
- `n in list` - test for element existence

```
  a = [1,2,3,4,5]
  a[0]                  # First element
1
  a[1:3]                # a[1] and a[2]
[2, 3]

  a[3] = 99             # lists are mutable

  max(a)
99

  a.sort()              # sort in place
[1, 2, 3, 5, 99]

  5 in a                # is 5 in a?
True
```

Lists: ranges and loops

```
for <var> in <iterable>:
    <do something with var>

# example:
a = [1,2,3,4,5]
for i in a:
     print(i)

# range
for i in range(1,10):
    print(i)

# create a list from a range

a = list(range(1,10,2))
```

do something with each list element with '`for`'

- the index variable is set to each element in the list in turn

`range()` creates a sequence

`range(start, end, step)`

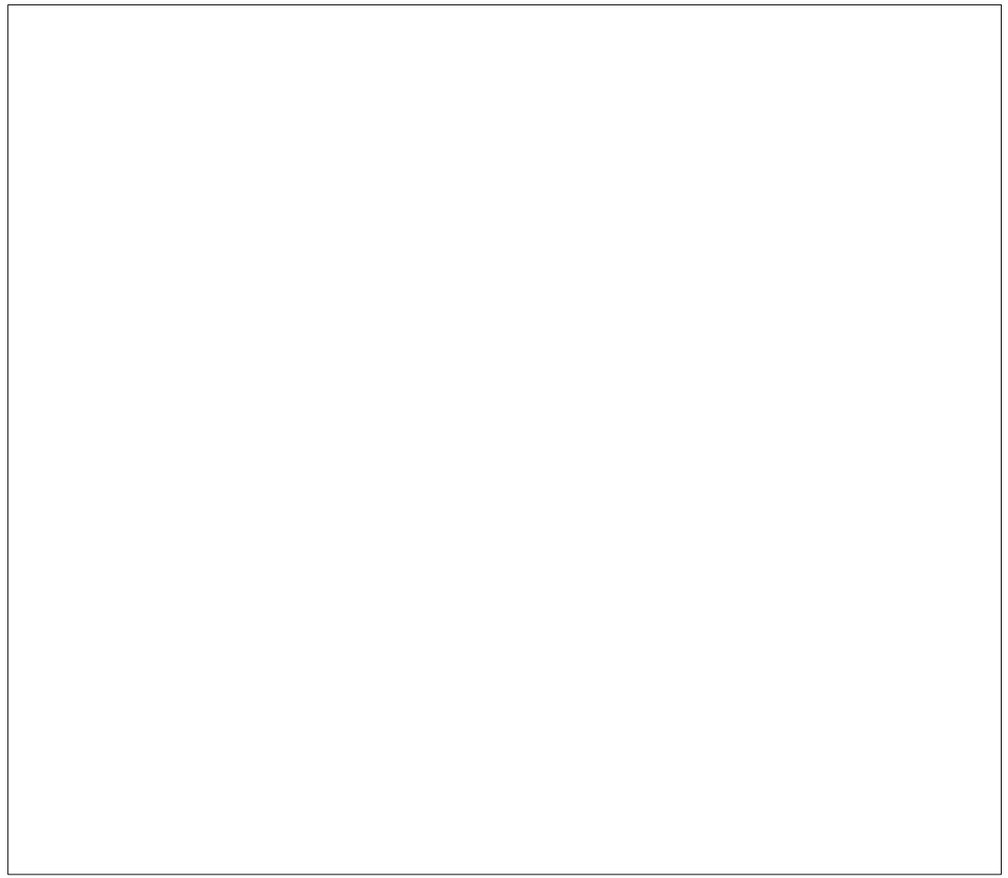
- creates sequence

  `start` - first element

  `end` - last element +1

  `step` - step size (optional)

Mini exercise: a list of squares



Create a list of squares of 1 to 10:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- Hints:
  - list.append(element)
  - range(start, end)
- write code in a file “squares.py”
- Test either from command line
  - `python3 squares.py`
- or within ipython:
  - `%run squares.py`

Mini exercise: a list of squares

```
# Create empty list


# Set i to each value in 1-10


    # calculate i**2, append
    # to the list


# After the loop, print list
```

Create a list of squares of 1 to 10:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- Hints:
  - list.append(element)
  - range(start, end)
- write code in a file “squares.py”
- Test either from command line
  - `python3 squares.py`
- or within ipython:
  - `%run squares.py`

Mini exercise: a list of squares

```
# Create empty list
squares = []

# Set i to each value in 1-10
for i in range(1,11):

    # calculate i**2, append
    # to the list
    squares.append(i**2)

# After the loop, print list
print(squares)
```

Create a list of squares of 1 to 10:

`[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]`

- Hints:
  - list.append(element)
  - range(start, end)
- write code in a file “squares.py”
- Test either from command line
  - `python3 squares.py`
- or within ipython:
  - `%run squares.py`

List Comprehensions

A common pattern:

```
lst = []
for x in <iterable>:
  if <condition>:
    lst.append(<x comp>)
```

For example, our exercise:

```
squares = []
for x in range(1,11):
  # if <condition>:
    squares.append(x**2)
```

List Comprehensions

A common pattern:

```
lst = []
for x in <iterable>:
  if <condition>:
    lst.append(<x comp>)
```

Write it shorter as:

```
lst = [<x comp> for x in <iter> if <cond>]
```

For example, our exercise:

```
squares = []
for x in range(1,11):
  # if <condition>:
    squares.append(x**2)
```

List Comprehensions

A common pattern:

```
lst = []
for x in <iterable>:
  if <condition>:
    lst.append(<x comp>)
```

Write it shorter as:

```
lst = [<x comp> for x in <iter> if <cond>]
```

For example, our exercise:

```
squares = []
for x in range(1,11):
  # if <condition>:
    squares.append(x**2)
```

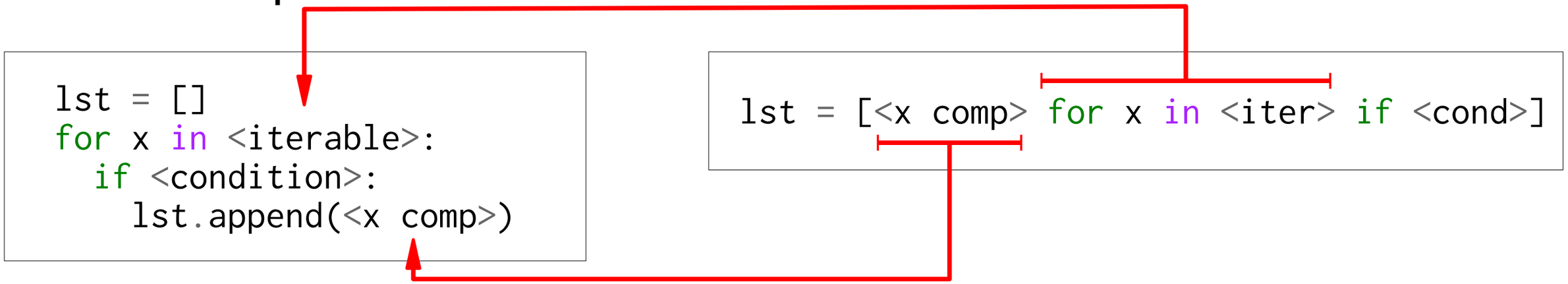
List Comprehensions

A common pattern:

```
lst = []
for x in <iterable>:
  if <condition>:
    lst.append(<x comp>)
```

Write it shorter as:

```
lst = [<x comp> for x in <iter> if <cond>]
```



For example, our exercise:

```
squares = []
for x in range(1,11):
  # if <condition>:
    squares.append(x**2)
```

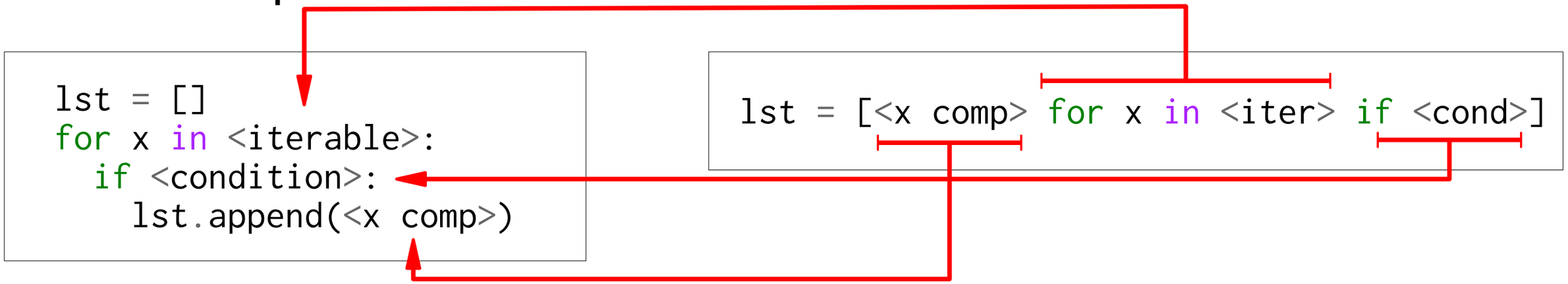
List Comprehensions

A common pattern:

```
lst = []
for x in <iterable>:
  if <condition>:
    lst.append(<x comp>)
```

Write it shorter as:

```
lst = [<x comp> for x in <iter> if <cond>]
```



For example, our exercise:

```
squares = []
for x in range(1,11):
  # if <condition>:
    squares.append(x**2)
```

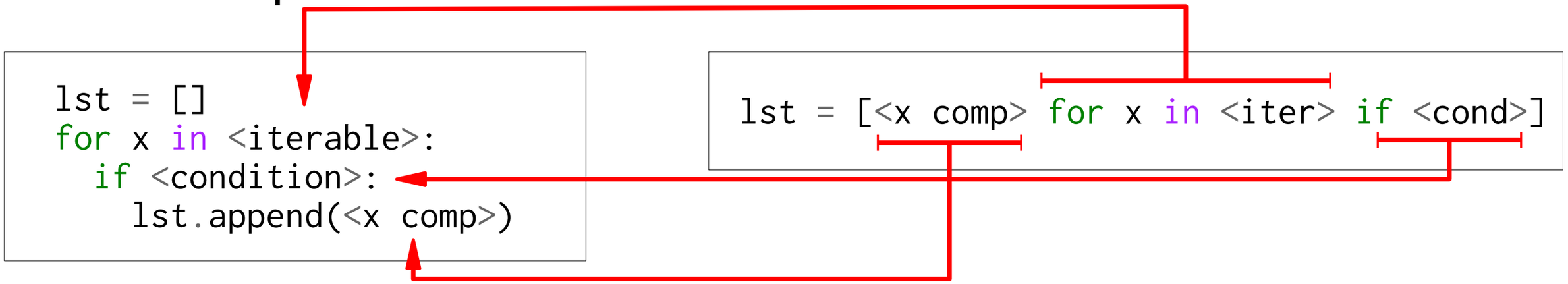
List Comprehensions

A common pattern:

```
lst = []
for x in <iterable>:
  if <condition>:
    lst.append(<x comp>)
```

Write it shorter as:

```
lst = [<x comp> for x in <iter> if <cond>]
```



For example, our exercise:

```
squares = []
for x in range(1,11):
  # if <condition>:
    squares.append(x**2)
```

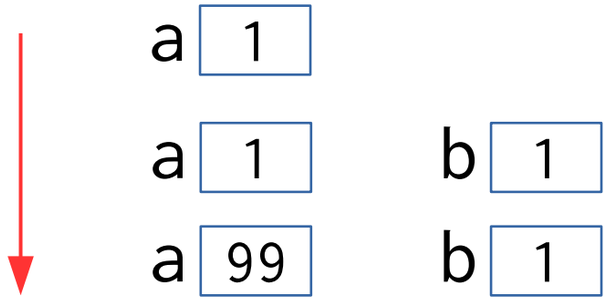
Our exercise again:

```
squares = [x**2 for x in range(1,11)]
```

Lists and simple values

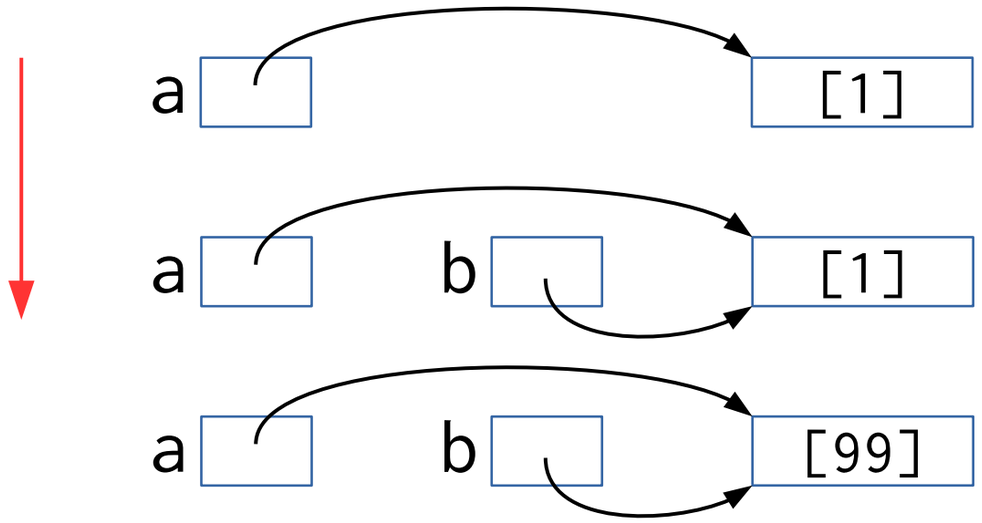
variables *contain* simple values:

```
  a = 1          # a == 1
  b = a          # b == 1
  a = 99         # a == 99, b == 1
  print(a,b)
99, 1
```



variables contain *references to* lists:

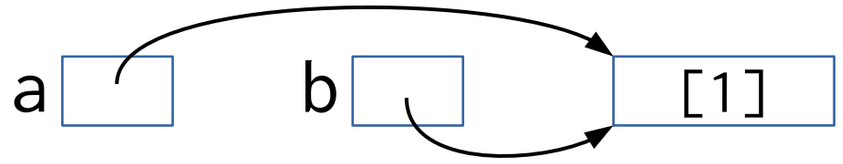
```
  a = [1]        # a == [1]
  b = a          # b == [1]
  a[0] = 99      # a == [99], b == [99]
  print(a,b)
[99], [99]
```



Lists and simple values

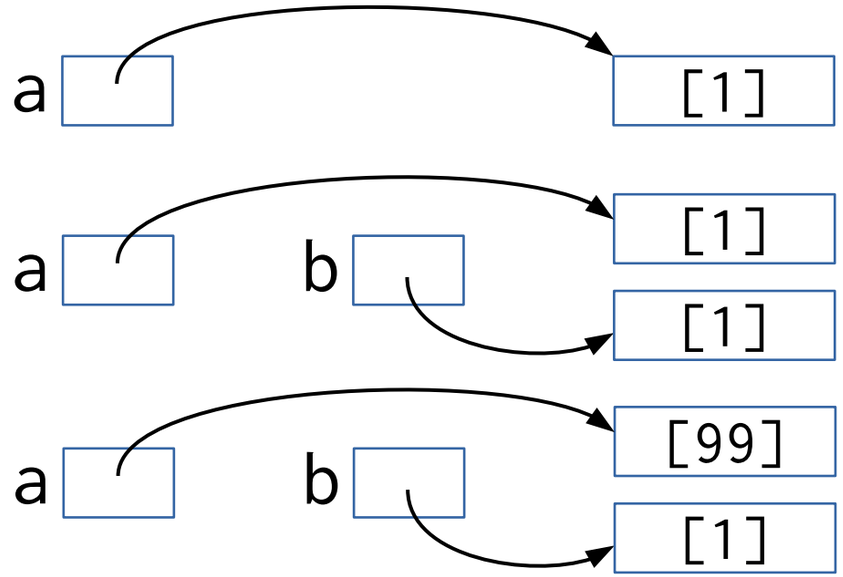
Without using `.copy()`

```
  a = [1]       # a == [1]
  b = a         # b == [1]
  a[0] = 99     # a == [99], b == [99]
  print(a,b)
[99], [99]
```



With `.copy()`

```
  a = [1]         # a == [1]
  b = a.copy()    # b == [1]
  a[0] = 99       # a == [99], b == [1]
  print(a,b)
[99], [1]
```



Functions

functions *encapsulate* code

```
def <function name>(params):
    <code block>
```

Docstring

- A string after the function definition with documentation

Parameters

- parameters can have default values
- you can name parameters when calling a function

`return` sets the return value

```
def line(x, a=0, b=0):
    '''line(x, a=0, b=0)
    calculate ax+b'''

    return a*x+b


line(2, 0.5, 1)    # x=2, a=0.5, b=1
2.0

line(2)            # a,b=0 default
0

line(2, b=5)       # a=0, b=5
5
```

Functions

```
  def func(x):
      a = 1         # local a
      x +=1         # local parameter x
      print("a:",a,"x:",x)


  a=99              # global a
  x=11              # global x
  func(x)
a: 1 x: 12          # local a and x

  print("a:",a,"x:",x)
a: 99 x: 11         # global a and x
```

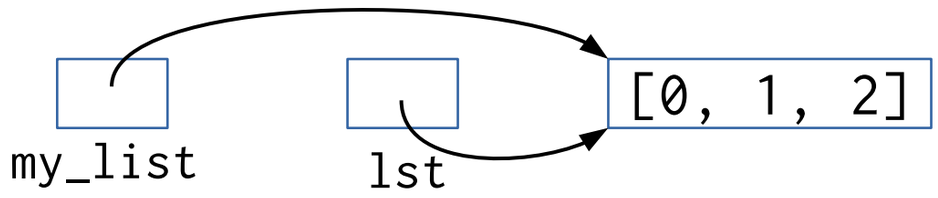
Parameters and variables

- parameters and local variables are *local* to the function.
- Parameters copy the value, not the parameter itself

(you *can* use global variables in a function. But please don't.)

Functions

```
  def listfunc(lst):
      lst[0] = 99       # local list
      print("list:", lst)


  my_list = [0,1,2]     # global list
  listfunc(my_list)
list: [99, 1, 2]

  print("list", my_list)
list: [99, 1, 2]
```



Parameters and variables

- parameters and local variables are *local* to the function.
- Parameters copy the value, not the parameter itself

- you transfer a *reference* to lists

  → functions can change the contents of the list

Functions

Return values

- return any value with '`return`'
- can put `return` anywhere in the code

- A function without return returns `None`
- You can use '`None`' yourself to signify the lack of a value, etc.

```
  def print_double(x):
      print(x, x)

  a = print_double(3)
3 3

  print(a)
None

  def print_if_exist(x=None):
      if x==None:
          print("no value")
      else:
          print(x)
```

Other structures

tuples

- like lists, but immutable
  → faster, take less memory

dictionaries

- key - value pairs.
- key : immutable (string, numeric)
  value : any kind of value
- No intrinsic order
- very fast, flexible

```
  my_tuple = (1, 2, 3)
  my_tuple[1]
2

 my_tuple[1] = 99          # Error

  my_dict = {'name': 'Janne',
             'age' : 22}

  my_dict['name']
Janne

  my_dict['numbers']=[1,2,3]
  for k,v in my_dict.items():
      print('{}: {}'.format(k,v))
age: 22
numbers: [1, 2, 3]
name: Janne
```

File I/O

open file for reading or writing:

```
with open(<fname>, 'rw') as <var>:
    <code block>
```

`readline()` - read one line
`readlines()` - read list of all lines in file
`write(l)` - write a string
`writelines(l)` - write all lines in list

```
# open file for reading
with open('filename','r') as f:
    # read all lines at once
    lines = f.readlines()

    # read one line at a time
    for line in f:
        # do something w. each line

# write all lines to filename
with open('filename','w') as f:
    f.writelines(lines)

# f.writeline(l) writes one line
```

Next Session:

We will try out four useful things right on the Deigo cluster:

- Use python modules
- Install python modules for your own use
- write a program that converts a file (a Fasta file) from one format to another
- Write a program that creates a Slurm job script, then submits it as a job.