

F21SC Industrial Programming: Python Introduction & Control Flow

Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh



Semester 1 — 2019/20

⁰No proprietary software has been used in producing these slides

Contents

- 1 Python Overview
- 2 Getting started with Python
- 3 Control structures
- 4 Functions
- 5 Classes
- 6 Exceptions
- 7 Iterators and Generators
- 8 Overloading
- 9 More about Types and Classes
- 10 Decorating Functions
- 11 Interpretation and Compilation
- 12 Functional Programming in Python
- 13 Libraries

Online Resources

- `www.python.org`: official website
- Course mostly based on *Guido van Rossum's* tutorial.
- For textbooks in Python introductions see the end of this slideset.
- Stable version: 3.8 (Oct 2019)
- Implemented in C (CPython)

Python

- Python is named after *Monty Python's Flying Circus*
- Python is an object-oriented language focussing on rapid prototyping
- Python is a scripting language
- Python features an elegant language design, is easy to learn and comprehend
- Open source
- Highly portable
- First version was made available 1990
- Current stable version is 3.8 (Oct 2019)

Python 3 vs Python 2

We will use Python 3, which offers several important new concepts over Python 2.

If you find Python 2 code samples, they might not run with python3. There is a tool `python3-2to3` which tells you what to change (and it works in most cases). The most common issues are

- In Python 3, `print` is treated as any other function, especially you need to use parentheses as in write `print(x)` NOT `print x`
- Focus on iterators: pattern-like functions (e.g. `map`) now return iterators, i.e. a handle used to perform iteration, rather than a data structure.

For details check:

<https://www.python.org/downloads/release/python-363/>

Runtime behaviour

- Python source code is compiled to byte-code, which is then interpreted
- Compilation is performed transparently
- Automatic memory management using *reference counting* based garbage collection
- No uncontrolled crash (*as in seg faults*)

Language features

- Everything is an object (pure object-oriented design)
- Features classes and multiple inheritance
- Higher-order functions (similar to Scheme)
- Dynamic typing and polymorphism
- Exceptions as in Java
- Static scoping and modules
- Operator overloading
- Block structure with semantic-bearing indentation (“off-side rule” as in Haskell)

Data types

- Numbers: `int`, `long`, `float`, `complex`
- Strings (similar to Java)
- Tuples, Lists, Dictionaries
- Add-on modules can define new data-types
- Can model arbitrary data-structures using classes

Why Python?

- Code **2 – 10×** shorter than C#, C++, Java
- Code is easy to comprehend
- Encourages *rapid prototyping*
- Good for *web scripting*
- Scientific applications (numerical computation, natural language processing, data visualisation, etc)
- Python is increasingly used at US universities as a *starting language*
- Rich libraries for XML, Databases, Graphics, etc.
- *Web content management* (Zope/Plone)
- GNU Mailman
- JPython

Python vs. other languages

- Very active community
- A lot of good libraries
- Increasingly used in teaching (MIT, Berkeley, etc)
- Good online teaching material, e.g. [Online Python Tutor](#)
- Picks up many advanced language features from other languages (e.g. Haskell)

Python Textbooks (Advanced)



Mark Lutz, *“Programming Python.”*

O'Reilly Media; 4 edition (10 Jan 2011). ISBN-10: 0596158106.

Good textbook for more experienced programmers. Detailed coverage of libraries.



David M. Beazley, *“Python Essential Reference.”*

Addison Wesley; 4 edition (9 July 2009). ISBN-10: 0672329786.

Detailed reference guide to Python and libraries.



Alex Martelli, Anna Ravenscroft, Steve Holden, *“Python in a Nutshell.”*

O'Reilly Media; 3rd edition (May 2017). ISBN-13: 978-1449392925

Concise summary of Python language and libraries.

Python Textbooks (Beginner)



Mark Lutz, *“Learning Python.”*,

5th edition, O’Reilly, 2013. ISBN-10: 1449355730

Introduction to Python, assuming little programming experience.



John Guttag. *“Introduction to Computation and Programming Using Python.”*, MIT Press, 2013. ISBN: 9780262519632.

Doesn’t assume any programming background.



Timothy Budd. *“Exploring Python.”*,

McGraw-Hill Science, 2009. ISBN: 9780073523378.

Exploring Python provides an accessible and reliable introduction into programming with the Python language.

Python Textbooks (Beginner)

 Zed A. Shaw. *“Learn Python the Hard Way.”*,

Heavily exercise-based introduction to programming. Good on-line material.

 Michael Dawson, *“Python Programming for the Absolute Beginner.”*,

3rd edition, Cengage Learning PTR, 2010. ISBN-10: 1435455002
Good introduction for beginners. Slightly dated. Teaches the principles of programming through simple game creation.

 Tony Gaddis, *“Starting Out with Python.”*,

Pearson New International Edition, 2013. ISBN-10: 1292025913
Good introduction for beginners..

Python Textbooks (Beginner)

Online resources:

- [How to Think Like a Computer Scientist.](#)
- [An Introduction to Python.](#)
- [Dive into Python 3.](#)
- [Google's Python Class.](#)
- [Main Python web page.](#)

For this course:

- Main course information page:
http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/index_new.html.
- Python sample code:
http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/Samples/python_sample
- FAQs:
<http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/faq.html#python>

Python Textbooks (Beginner)

Online resources:

- [How to Think Like a Computer Scientist.](#)
- [An Introduction to Python.](#)
- [Dive into Python 3.](#)
- [Google's Python Class.](#)
- [Main Python web page.](#)

For this course:

- Main course information page:
http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/index_new.html.
- Python sample code:
http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/Samples/python_sample
- FAQs:
<http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/faq.html#python>

Launching Python

- Interactive Python shell: `python`
- Exit with *eof* (Unix: Ctrl-D, Windows: Ctrl-Z)

- Or: `import sys; sys.exit()`

- Execute a script: `python myfile.py`

```
python3 ..python-args.. script.py ..script-args..
```

- Evaluate a Python expression

```
python3 -c "print (5*6*7) "
```

```
python3 -c "import sys; print (sys.maxint) "
```

```
python3 -c "import sys; print (sys.argv)" 1 2 3 4
```

- Executable Python script

```
#!/usr/bin/env python3
```

```
# -*- coding: iso-8859-15 -*-
```

Integer Arithmetic

- `>>>` is the Python prompt, asking for input

```
>>> 2+2    # A comment on the same line as code.
```

```
4
```

```
>>> # A comment; Python asks for a continuation ...
```

```
... 2+2
```

```
4
```

```
>>> (50-5*6) / 4
```

```
5.0
```

```
>>> # Use // for integer division (returns floor):
```

```
... 7//3
```

```
2
```

```
>>> 7//-3
```

```
-3
```

Arbitrary precision integers

- `int` represents signed integers (32/64 Bit).

```
>>> import sys; sys.maxint
2147483647
```

- `long` represents arbitrary precision integers.

```
>>> sys.maxint + 1
2147483648L
>>> 2 ** 100
1267650600228229401496703205376L
```

- Conversion: [`“_”` is a place-holder for an *absent* value.]

```
>>> - 2 ** 31
-2147483648L
>>> int(_)
-2147483648
```

Assignment

- Variables don't have to be declared (scripting language).

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

- Parallel assignments:

```
>>> width, height = height, width + height
```

- Short-hand notation for parallel assignments:

```
>>> x = y = z = 0 # Zero x, y and z
>>> x
0
>>> z
0
```

Floating-point numbers

- Arithmetic operations are overloaded.
- Integers will be converted on demand:

```
>>> 3 * 3.75 / .5
```

```
22.5
```

```
>>> 7. / 2
```

```
3.5
```

```
>>> float(7) / 2
```

```
3.5
```

- Exponent notation: `1e0` `1.0e+1` `1e-1` `.1e-2`
- Typically with 53 bit precision (as `double` in C).

```
>>> 1e-323
```

```
9.8813129168249309e-324
```

```
>>> 1e-324
```

```
0.0
```

Further arithmetic operations

- Remainder:

```
>>> 4 % 3
```

```
1
```

```
>>> -4 % 3
```

```
2
```

```
>>> 4 % -3
```

```
-2
```

```
>>> -4 % -3
```

```
-1
```

```
>>> 3.9 % 1.3
```

```
1.2999999999999998
```

- Division and Floor:

```
>>> 7.0 // 4.4
```

```
1.0
```

Complex Numbers

- Imaginary numbers have the suffix `j`.

```
>>> 1j * complex(0,1)
(-1+0j)
>>> complex(-1,0) ** 0.5
(6.1230317691118863e-17+1j)
```

- Real- and imaginary components:

```
>>> a=1.5+0.5j
>>> a.real + a.imag
2.0
```

- Absolute value is also defined on `complex`.

```
>>> abs(3 + 4j)
5.0
```

Bit-operations

- Left- (<<) and right-shift (>>)

```
>>> 1 << 16
65536
```

- Bitwise and (&), or (|), xor (^) and negation (~).

```
>>> 1000 & 0377
232
>>> 0x7531 | 0x8ace
65535
>>> ~0
-1
>>> 0123 ^ 0123
0
```

Strings

- Type: `str`.
- Single- and double-quotes can be used

Input	Output
-----	-----
<code>'Python tutorial'</code>	<code>'Python tutorial'</code>
<code>'doesn\'t'</code>	<code>"doesn't"</code>
<code>"doesn't"</code>	<code>"doesn't"</code>
<code>'"Yes," he said.'</code>	<code>'"Yes," he said.'</code>
<code>"\"Yes,\" he said."</code>	<code>'"Yes," he said.'</code>
<code>'"Isn\'t," she said.'</code>	<code>'"Isn\'t," she said.'</code>

Escape-Sequences

<code>\\</code>	backslash
<code>\'</code>	single quote
<code>\"</code>	double quote
<code>\t</code>	tab
<code>\n</code>	newline
<code>\r</code>	carriage return
<code>\b</code>	backspace

Multi-line string constants

- The expression

```
print ("This is a rather long string containing\n\
several lines of text as you would do in C.\n\
    Whitespace at the beginning of the line is\
significant.")
```

- displays this text

```
This is a rather long string containing
several lines of text as you would do in C.
    Whitespace at the beginning of the line is sign
```

Triple-quote

- Multi-line string including line-breaks:

```
print ("""
Usage: thingy [OPTIONS]
    -h                Display this usage message
    -H hostname       Hostname to connect to
""")
```

- gives

```
Usage: thingy [OPTIONS]
    -h                Display this usage message
    -H hostname       Hostname to connect to
```

Raw strings

- An `r` as prefix preserves all escape-sequences.

```
>>> print ("Hello! \n\"How are you?\")
```

```
Hello!
```

```
"How are you?"
```

```
>>> print (r"Hello! \n\"How are you?\")
```

```
Hello! \n\"How are you?"
```

- Raw strings also have type `str`.

```
>>> type ("\n")
```

```
<type 'str'>
```

```
>>> type (r"\n")
```

```
<type 'str'>
```

Unicode

- Unicode-strings (own type) start with `u`.

```
>>> print ("a\u0020b")
a b
>>> "\xf6"
"ö"
>>> type (_)
<type 'unicode'>
```

- Standard strings are converted to unicode-strings on demand:

```
>>> "this " + "\u00f6" + " umlaut"
'this ö umlaut'
>>> print _
this ö umlaut
```

String operations

● "hello"+"world"	"helloworld"	# concat.
"hello"*3	"hellohellohello"	# repetition
"hello"[0]	"h"	# indexing
"hello"[-1]	"o"	# (from end)
"hello"[1:4]	"ello"	# slicing
len("hello")	5	# size
"hello" < "jello"	True	# comparison
"e" in "hello"	True	# search

Lists

- Lists are *mutable arrays*.

```
a = [99, "bottles of beer", ["on", "the", "wall"]]
```

- String operations also work on lists.

```
a+b, a*3, a[0], a[-1], a[1:], len(a)
```

- Elements and segments can be modified.

```
a[0] = 98
```

```
a[1:2] = ["bottles", "of", "beer"]  
# -> [98, "bottles", "of", "beer",  
      ["on", "the", "wall"]]
```

```
del a[-1] # -> [98, "bottles", "of", "beer"]
```

More list operations

```
>>> a = range(5)           # [0, 1, 2, 3, 4]
>>> a.append(5)             # [0, 1, 2, 3, 4, 5]
>>> a.pop()                 # [0, 1, 2, 3, 4]
5
>>> a.insert(0, 42)         # [42, 0, 1, 2, 3, 4]
>>> a.pop(0)                # [0, 1, 2, 3, 4]
42
>>> a.reverse()             # [4, 3, 2, 1, 0]
>>> a.sort()                # [0, 1, 2, 3, 4]
```

N.B.: Use `append` for push.

While

- Print all Fibonacci numbers up to 100 (interactive):

```
>>> a, b = 0, 1
>>> while b <= 100:
...     print (b)
...     a, b = b, a+b
... 
```

- Comparison operators: == < > <= >= !=
- **NB:** Indentation carries semantics in Python:
 - ▶ Indentation starts a block
 - ▶ De-indentation ends a block
- Or:

```
>>> a, b = 0, 1
>>> while b <= 100: print (b); a,b = b, a+b
... 
```

If

Example

```
x = int(input("Please enter an integer: "))
if x < 0:
    x = -1
    print('Sign is Minus')
elif x == 0:
    print('Sign is Zero')
elif x > 0:
    print('Sign is Plus')
else:
    print('Should never see that')
```

- **NB:** elif instead of else if to avoid further indentations.

For

- `for` iterates over a sequence (e.g. list, string)

Example

```
a = ['cat', 'window', 'defenestrate']
for x in a:
    print(x, len(x))
```

- **NB:** The iterated sequence must not be modified in the body of the loop! However, it's possible to create a copy, e.g. using segment notation.

```
for x in a[:]:
    if len(x) > 6: a.insert(0,x)
print (a)
```

- Results in

```
['defenestrate', 'cat', 'window', 'defenestrate']
```

Range function

- Iteration over a sequence of numbers can be simplified using the `range()` function:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

- Iteration over the indices of an array can be done like this:

```
a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
    print (i, a[i])
```

For-/While-loops: `break`, `continue`, `else`

- `break` (as in C), terminates the enclosing loop immediately.
- `continue` (as in C), jumps to the next iteration of the enclosing loop.
- The `else`-part of a loop will only be executed, if the loop hasn't been terminated using `break` construct.

Example

```
for n in range(2, 10):  
    for x in range(2, n):  
        if n % x == 0:  
            print (n, 'equals', x, '*', n//x)  
            break  
    else: # loop completed, no factor  
        print (n, 'is a prime number')
```

The empty expression

- The expression `pass` does nothing.

```
while True:
```

```
    pass # Busy-wait for keyboard interrupt
```

- This construct can be used, if an expression is syntactically required, but doesn't have to perform any work.

Procedures

- Procedures are defined using the key word `def`.

```
def fib(n):      # write Fibonacci series up to n
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while b < n:
        print (b)
        a, b = b, a+b
```

- Variables `n`, `a`, `b` are local.
- The return value is `None` (hence, it is a procedure rather than a function).

```
print (fib(10))
```

A procedure as an object

- Procedures are values in-themselves.

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

Call-by-value

- When passing arguments to functions, a **Call-by-value** discipline is used (as in C, C++, or C#).
- Assignment to parameters of a function are local.

```
def bla(l):  
    l = []
```

```
l = ['not', 'empty']  
bla(l)  
print(l)
```

- `l` is a reference to an object.
- The referenced object can be modified:

```
def exclamate(l):  
    l.append('!')
```

```
exclamate(l)  
print(l)
```

Global Variables

- The access to a global variable has to be explicitly declared.

```
def clear_l():  
    global l  
    l = []  
  
l = ['not', 'empty']  
clear_l()  
print(l)
```



Global Variables

- The access to a global variable has to be explicitly declared.

```
def clear_l():  
    global l  
    l = []
```

```
l = ['not', 'empty']  
clear_l()  
print(l)
```

- ...prints the *empty* list.

Return values

- The `return` construct immediately terminates the procedure.
- The `return ...value...` construct also returns a concrete result value.

```
def fib2(n):  
    """Return the Fibonacci series up to n."""  
    result = []  
    a, b = 0, 1  
    while b < n:  
        result.append(b)      # see below  
        a, b = b, a+b  
    return result
```

```
f100 = fib2(100)    # call it  
f100                # write the result
```

Default values for function parameters

- In a function definition, default values can be specified for parameters:

```
def ask(prompt, retries=4, complaint='Yes/no?'):  
    while True:  
        ok = raw_input(prompt)  
        if ok in ('y', 'ye', 'yes'): return True  
        if ok in ('n', 'no'): return False  
        retries -= 1  
        if retries < 0: raise IOError, 'refused'  
        print (complaint)
```

- When calling the function, some arguments can be omitted.

```
ask ("Continue (y/n)?", 3, "Yes or no, please!")  
ask ("Continue (y/n)?", 3)  
ask ("Continue (y/n)?")
```

Default values for function parameters (cont'd)

- Wrong:

```
ask ("Continue (y/n)?", "Yes or no, please!")  
ask ()
```

- Named arguments (*keyword arg*) are useful when using arguments with and without default values:

```
ask ("Continue (y/n)?", complaint="Yes or no?")  
ask (prompt="Continue (y/n)?")
```

- Wrong:

```
ask (prompt="Continue (y/n)?", 5)  
ask ("Yes/no?", prompt="Continue (y/n)?")
```

Evaluation of default values:

- Default values will be evaluated *only once*, when the function is *defined*:

```
i = 5
```

```
def f(arg=i):  
    print (arg)
```

```
i = 6
```

```
f()
```

- Which number will be printed?



Evaluation of default values:

- Default values will be evaluated *only once*, when the function is *defined*:

```
i = 5
```

```
def f(arg=i):  
    print (arg)
```

```
i = 6
```

```
f()
```

- Which number will be printed?
- ...prints 5.

Evaluation of default values

- Beware of mutable objects!

```
def f(a, L=[]):  
    L.append(a)  
    return L
```

```
print (f(1))  
print (f(2))
```

- ... prints [1] and [1, 2]. However:

```
def f(a, L=None):  
    if L is None:  
        L = []  
    L.append(a)  
    return L
```

- ... prints [1] and [2].

Argument lists

- Prefixing a parameter with `*` declares a parameter that can take an arbitrary number of values.

```
def fprintf(file, format, *args):  
    file.write(format % args)
```

- A list can be passed as individual arguments using `*` notation:

```
>>> args = [3, 6]  
>>> range(*args)  
[3, 4, 5]
```

Doc-strings

- The first expression in a function can be a string (as in elisp).

```
def my_function():  
    """Do nothing, but document it.  
  
    No, really, it doesn't do anything.  
    """  
    pass
```

- The first line typically contains usage information (starting with an upper-case letter, and terminated with a full stop).
- After that several more paragraphs can be added, explaining details of the usage information.
- This information can be accessed using `.__doc__` or `help` constructs.

```
my_function.__doc__    # return doc string  
help(my_function)     # print doc string
```

Anonymous Functions

- A function can be passed as an expression to another function:

```
>>> lambda x, y: x  
<function <lambda> at 0xb77900d4>
```

- This is a factory-pattern for a function incrementing a value:

```
def make_incrementor(n):  
    return lambda x: x + n
```

```
f = make_incrementor(42)  
f(0)  
f(1)
```

- Functions are compared using the address of their representation in memory:

```
>>> (lambda x: x) == (lambda x: x)  
False
```

Exercises

- Implement Euclid's greatest common divisor algorithm as a function over 2 int parameters.
- Implement matrix multiplication as a function taking 2 2-dimensional arrays as arguments.

More list operations

- **Modifiers:**

- ▶ `l.extend(l2)` means `l[len(l) :] = l2`, i.e. add `l2` to the end of the list `l`.
- ▶ `l.remove(x)` removes the first instance of `x` in `l`. Error, if `x not in l`.

- **Read-only:**

- ▶ `l.index(x)` returns the position of `x` in `l`. Error, if `x not in l`.
- ▶ `l.count(x)` returns the number of occurrences of `x` in `l`.
- ▶ `sorted(l)` returns a new list, which is the sorted version of `l`.
- ▶ `reversed(l)` returns an iterator, which lists the elements in `l` in reverse order.

Usage of lists

- Lists can be used to model a *stack*: `append` and `pop()`.
- Lists can be used to model a *queue*: `append` and `pop(0)`.

Higher-order functions on lists

- `filter(test, sequence)` returns a sequence, whose elements are those of `sequence` that fulfill the predicate `test`.
E.g.

```
filter(lambda x: x % 2 == 0, range(10))
```

- `map(f, sequence)` applies the function `f` to every element of `sequence` and returns it as a new sequence.

```
map(lambda x: x*x*x, range(10))
```

```
map(lambda x,y: x+y, range(1,51), range(100,50,-1))
```

- `reduce(f, [a1,a2,a3,...,an])` computes
`f(...f(f(a1,a2),a3),...,an)`

```
reduce(lambda x,y:x*y, range(1,11))
```

- `reduce(f, [a1,a2,...,an], e)` computes
`f(...f(f(e,a1),a2),...,an)`

List comprehensions

- More readable notation for combinations of `map` and `filter`.
- Motivated by *set comprehensions* in mathematical notation.
- `[ e(x,y) for x in seq1 if p(x) for y in seq2 ]`

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
```

Deletion

- Deletion of (parts of) a list:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

- Deletion of variables:

```
>>> del a
```

Tuples

```
• >>> 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))
>>> x, y, z = t
>>> empty = ()
>>> singleton = 'hello',      # trailing comma
```

Sets

- `set(l)` generates a set, formed out of the elements in the list `l`.
- `list(s)` generates a list, formed out of the elements in the set `s`.
- `x in s` tests for set membership
- Operations: `-` (difference), `|` (union), `&` (intersection), `^` (xor).
- `for v in s` iterates over the set (sorted!).

Dictionaries

- Dictionaries are finite maps, *hash maps*, *associative arrays*.
- They represent unordered sets of (key, value) pairs.
- Every key may only occur once.
- Generated using the notation:

```
{ key1 : value1, ..., keyn : valuen } or
```

```
>>> tel = dict([('guido', 4127), ('jack', 4098)])
```

```
{'jack': 4098, 'guido': 4127}
```
- Access to elements is always through the key: `tel['jack']`.
- Insertion and substitution is done using assignment notation:
`tel['me'] = 1234.`
- Deletion: `del tel['me']`.
- `tel.keys()` returns all key values. `tel.has_key('guido')` returns a boolean, indicating whether the key exists.

Dictionaries

- The Python implementation uses dictionaries internally, e.g. to list all names exported by a module, or for the symbol table of the interpreter.

- Iteration over a dictionary:

```
for k, v in tel.items():  
    print (k, v)
```

- Named arguments:

```
def fun(arg, *args, **keyArgs): ...
```

```
fun (1, 2, 3, opt1=4, opt2=5)
```

- This binds `arg = 1` and `args = [2,3]` and `keyArgs = {opt1:4, opt2:5}`.

Loop techniques

- Here are some useful patterns involving loops over dictionaries.
- Simultaneous iteration over both keys and elements of a dictionary:

```
l = ['tic', 'tac', 'toe']  
for i, v in enumerate(l):  
    print (i, v)
```

- Simultaneous iteration over two or more sequences:

```
for i, v in zip(range(len(l)), l):  
    print (i, v)
```

- Iteration in sorted and reversed order:

```
for v in reversed(sorted(l)):  
    print (v)
```

Booleans

- 0, '', [], None, etc. are interpreted as `False`.
- All other values are interpreted as `True` (also functions!).
- `is` checks for object identity: `[] == []` is true, but `[] is []` isn't. `5 is 5` is true.
- Comparisons can be chained like this: `a < b == c > d`.
- The boolean operators `not`, `and`, or `or` are *short-cutting*.

```
def noisy(x): print (x); return x
```

```
a = noisy(True) or noisy(False)
```

- This technique can also be used with non-Boolean values:

```
>>> '' or 'you' or 'me'  
'you'
```

Comparison of sequences and other types

- Sequences are compared lexicographically, and in a nested way:

```
() < ('\x00',)
```

```
('a', (5, 3), 'c') < ('a', (6,) , 'a')
```

- NB:** The comparison of values of *different* types doesn't produce an error but returns an arbitrary value!

```
>>> "1" < 2
```

```
False
```

```
>>> () < ('\x00')
```

```
False
```

```
>>> [0] < (0,)
```

```
True
```

Modules

- Every Python file is a module.
- `import myMod` imports module `myMod`.
- The system searches in the current directory and in the `PYTHONPATH` environment variable.
- Access to the module-identifier `x` is done with `myMod.x` (both read and write access!).
- The code in the module is evaluated, when the module is imported the first time.
- Import into the main name-space can be done by

Example

```
from myMod import myFun
from yourMod import yourValue as myValue

myFun(myValue) # qualification not necessary
```

- **NB:** In general it is not advisable to do `from myMod import *`

Executing modules as scripts

- Using `__name__` the name of the module can be accessed.
- The name is `'__main__'` for main program:

Example

```
def fib(n): ...

if __name__ == '__main__':
    import sys
    fib(int(sys.argv[1]))
```

- Typical application: unittests.

Modules as values

- A module is an object.

```
>>> fib = __import__('fibonacci')
>>> fib
<module 'fibonacci' from 'fibonacci.py'>
>>> fib.fib(10)
1 1 2 3 5 8
```

- `fib.__name__` is the name of the module.
- `fib.__dict__` contains the defined names in the module.
- `dir(fib)` is the same as `fib.__dict__.keys()`.

Standard- und built-in modules

- See *Python Library Reference*, e.g. module `sys`.
- `sys.ps1` and `sys.ps2` contain the prompts.
- `sys.path` contains the module search-path.
- With `import __builtin__` it's possible to obtain the list of all built-in identifiers.

```
>>> import __builtin__  
>>> dir(__builtin__)
```

Packages

- A directory, that contains a (possibly empty) file `__init__.py`, is a package.
- Packages form a tree structure. Access is performed using the notation `packet1.packet2.module`.

Example

```
import packet.subpacket.module
print (packet.subpacket.module.__name__)

from packet.subpacket import module
print (module.__name__)
```

- If a package `packet/subpacket/__init__.py` contains the expression `__all__ = ["module1", "module2"]`, then it's possible to import both modules using `from packet.subpacket import *`

Output formatting

- `str(v)` generates a “machine-readable” string representation of `v`
- `repr(v)` generates a representation that is readable to the interpreter. Strings are escaped where necessary.
- `s.rjust(n)` fills the string, from the left hand side, with space characters to the total size of `n`.
- `s.ljust(n)` and `s.center(n)`, analogously.
- `s.zfill(n)` inserts zeros to the number `s` in its string representation.
- `'-3.14'.zfill(8)` yields `'%08.2f' % -3.14`.
- Dictionary-Formating:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098 }
```

```
>>> print ('Jack: %(Jack)d; Sjoerd: %(Sjoerd)d' % table)
Jack: 4098; Sjoerd: 4127
```

File I/O

- Standard-output is `sys.stdout`.
- `f = open(filename, mode)` creates a file-object `f`, referring to `filename`.
- Access modi are: `'r'`, `'w'`, `'a'`, `'r+'` (read, write, append, read-write) plus suffix `b` (binary).
- `f.read()` returns the entire contents of the file as a string.
`f.read(n)` reads the next `n` bytes.
- `f.readline()` reads the next line, terminated with `'\\n'`. Empty string if at the end of the file.
- `f.readlines()` returns a list of all lines.
- Iteration over all lines:

```
for line in f: print (l)
```

Writing and moving

- `f.write(s)` writes the string `s`.
- `f.seek(offset, 0)` moves to position `seek` (counting from the start of the file).
- `f.seek(offset, 1)` moves to position `seek` (counting from the current position).
- `f.seek(offset, 2)` moves to position `seek` (counting from the end of the file).
- `f.close()` closes the file.

Pickling

- Arbitrary objects can be written to a file.
- This involves serialisation, or “pickling”, of the data in memory.
- Module `pickle` provides this functionality.
- `pickle.dump(x, f)` turns `x` into a string and writes it to file `f`.
- `x = pickle.load(f)` reads `x` from the file `f`.

Saving structured data with JSON

- JSON (JavaScript Object Notation) is a popular, light-weight data exchange format.
- Many languages support this format, thus it's useful for data exchange across systems.
- It is much lighter weight than XML, and thus easier to use.
- `json.dump(x, f)` turns `x` into a string in JSON format and writes it to file `f`.
- `x = json.load(f)` reads `x` from the file `f`, assuming JSON format.
- For detail on the JSON format see: <http://json.org/>

JSON Example

Example

```
tel = dict([('guido', 4127), ('jack', 4098)])
ppTelDict(tel)

# write dictionary to a file in JSON format
json.dump(tel, fp=open(jfile,'w'), indent=2)
print("Data has been written to file ", jfile);

# read file in JSON format and turn it into a dictionary
tel_new = json.loads(open(jfile,'r').read())
ppTelDict(tel_new)

# test a lookup
the_name = "Billy"
printNoOf(the_name, tel_new);
```

Numerical Computation using the `numpy` library

- `numpy` provides a powerful library of mathematical/scientific operations
- Specifically it provides
 - ▶ a powerful N-dimensional array object
 - ▶ sophisticated (broadcasting) functions
 - ▶ tools for integrating C/C++ and Fortran code
 - ▶ useful linear algebra, Fourier transform, and random number capabilities
- For details see: <http://www.numpy.org/>

Numerical Computation Example: numpy

Example

```
import numpy as np
m1 = np.array([ [1,2,3],
                [7,3,4] ]); # fixed test input
# m1 = np.zeros((4,3),int); # initialise a matrix
r1 = np.ndim(m1);          # get the number of dimensions
m, p = np.shape(m1);       # no. of rows in m1 and no. of
# use range(0,4) to generate all indices
# use m1[i][j] to lookup a matrix element

print("Matrix m1 is an ", r1, "-dimensional matrix, of
```