Part VIII

Selected Topics

Potential Topics for This Part - Poll of WED

recommended

- 11: regular expressions
- 13: exceptions
- 12: unit tests

options

- 2: logging
- 9: anonymous functions (lambda) and `map` & `filter`
- 4: module `time`
- 12: module `argparse`
- 6: Jupyter notebook
- 1: function decorators
- 11: advanced plotting
- 6: extending Python using C/C++
- 8: faster code

Regular Expressions

Pattern Syntax

- Regular expressions should always be placed in a raw string
- E.g. `r'([^.]+\.(.*))'`

`.`	match any character but newline
`^`	match start of the string
`$`	match end of the string
`*`	match previous expression zero or more times (greedy)
`+`	match previous expression one or more times (greedy)
`?`	match previous expression zero or one time (greedy)
`*?, +?, ??`	non-greedy versions of `*, +, ?`
`{m}`	match previous expression m times
`{m,n}`	match previous expression m to n times (greedy)
`{m,n}?`	non-greedy version of `{m,n}`
`[...]`	match any character from the enclosed set
`[^...]`	match any character not in the set
`A|B`	matches A or B (both regular expressions)
`(...)`	stores the contents of the match inside the backets

Character Escape Sequences

- Characters with special meaning (`.`, `*`, `...`) can be used with their literal meaning by escaping them with a `\`, e.g. `\.`, `\*`
- Standard escape characters (`\n`, `\t`, `...`) work as expected, e.g. `r'\n+'` matches one or more newlines

`\`*`number*	match the text matched by group number (starting from 1)
`\d`	same as `[0-9]`
`\D`	same as `[^0-9]`
`\s`	matches any whitespace (same as `[\t\n\r\f\v]`
`\S`	matches nonwhitespace
`\w`	matches any alphanumeric character
`\W`	matches nonalphanumeric characters
`\A`	matches the start of a string
`\Z`	matches the end of a string

Regular Expressions (cont.)

- `findall(patt, string)` – find all non-overlapping matches
- `sub(patt, repl, string)` – replaces matches by repl

```
from re import *
regstr = r'\d*\.\d*|\d*'
s = "12.3/5/4.4;5.7;6"
findall(regstr,s)
regstr = r'(\d*\.\d*|\d*)'
sub(regstr, r'\1xxx', s)
```

- greedy vs. non-greedy

```
reg_greedy = r'<.*>'       #match as many chars as possible
s = "<H1>title</H1>"
findall(reg_greedy,s)      #whole '<H1>title</H1>' matched
reg_nongreedy = r'<.*?>' #match as few chars as possible
findall(reg_nongreedy,s) #only '<H1>' matched
```

Match Objects

- Some re functions do not return strings, but match objects (m)
- `match(patt, string)` returns MatchObject in case of match
- `search(patt, string)` searches for first match
- `finditer(patt, string)` like findall, but returns iterator
- `m.group([group1, group2]) returns subgroups of the match`

```
import re
s = "dates␣05/02/2010␣-␣05/14/2010"
patt = r'(\d+)/(\d+)/(\d+)'
for m in finditer(patt,s):
  print(m.group())
  print("%s-%s" % (m.group(2), m.group(3)))
```

Exceptions: Dealing with Errors

- Consider the user enters characters:

```
x = input("Please␣enter␣a␣number:␣")
print(float(x))
```

- Failed conversion raises an `ValueError` exception

```
Please enter a number: xyz
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for float(): xyz
```

- To deal with erroneous behaviour catch exception and handle it

Exceptions: Dealing with Errors (2)

- Syntax is

```
try:
    # do something
except ValueError as e:
    # handle ValueError
except Exception as e:
    # handle all other errors
else:
    # what to do if no error occured
finally:
    # code that should be executed in any case
```

Exceptions: Dealing with Errors (3)

- Back to the previous example

```
got_answer = False
while not got_answer:
    x = input("Please␣enter␣a␣number:␣")
    try:
        print(float(x))
        got_answer = True
    except ValueError as e:
        print("That␣wasn't␣a␣number")
        print(e  # do something with e)
```

- To raise your own errors (use one of many default types or create your own):

```
raise RuntimeError("Ooops!␣Something␣went␣wrong!")
raise IOError("File␣not␣existent...")
```

Taxonomy of Built-in Exceptions

```
BaseException                 # root of all exc.
  GeneratorExit
  KeyboardInterrupt           # Ctrl+C, e.g.
  SystemExit                  # Program Exit (sys.exit())
  Exception                   # Base for non-exit exc.
    StopIteration
    StandardError             # only Python 2.x
      ArithmeticError
        FloatingPointError
        ZeroDevisionError
      AssertionError
      AttributeError
      EnvironmentError
        IOError               # open("file.txt"), e.g.
        OSError
      EOFError
      ImportError             # modul not found, e.g.
```

```
LookupError
  IndexError            # tupels/lists, e.g.
  KeyError              # dictionaries, e.g.
MemoryError
NameError               # name not found
UnboundedLocalError
ReferenceError
RuntimeError
NotImplementedError
SyntaxError
  IndentationError
    TabError
SystemError
TypeError               # type error (2 + "3")
ValueError              # value errorr (float("three"))
  UnicodeError
    UnicodeDecodeError
    UnicodeEncodeError
    UnicodeTranslateError
```

Throw Built-in Exceptions

```
def readfloat1():
  while True:
    try:
      a = input("Number␣between␣0.0␣and␣1.0:␣")
      a = float(a)
      if(a<0.0 or a>1.0):
        raise ValueError("Number␣not␣in␣[0.0;␣1.0]")
    except ValueError as e:
      print("Error:␣%s" % e)
    else:
      return a
```

- At `a = float(a)`, an error is thrown automatically; the rest of the try Block is not being executed
- If `a` convertable, the value range is checked and a new exception is possibly thrown

Unit Tests – Motivation

automated testing

- (huge) pros:
  - avoids manual tests
  - helps to keep clean code
  - extremely important if more than 1 developer!
- (small) cons:
  - introduces additional code
  - who tests the tests?
- execution of your test plan
  - which parts of application/code shall be tested?
  - in which order?
  - what are the expected responses?
- Frameworks: exist for almost any programming language

Unit Tests – How to use it

- different packages available; most popular:
  - `unittest`
  - `nose2`
  - `pytest`
- We will use `unittest`:

```
import unittest

class MyTestClass(unittest.TestCase):

    def myFirstTestFunction(self):
        self.assertEqual( 3*4, 12)

    def mySecondTestFunction(self):
        self.assertEqual( 'a'*3, 'aaa')

unittest.main()
```

Unit Tests – Types of Assertions

Available assertions in `unittest`:

Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

source: `https://docs.python.org/3/library/unittest.html`

Unit Tests – Types of Assertions (2)

Available assertions in `unittest`:

Method	Checks that	New in
`assertAlmostEqual(a, b)`	`round(a-b, 7) == 0`	
`assertNotAlmostEqual(a, b)`	`round(a-b, 7) != 0`	
`assertGreater(a, b)`	`a > b`	3.1
`assertGreaterEqual(a, b)`	`a >= b`	3.1
`assertLess(a, b)`	`a < b`	3.1
`assertLessEqual(a, b)`	`a <= b`	3.1
`assertRegex(s, r)`	`r.search(s)`	3.1
`assertNotRegex(s, r)`	`not r.search(s)`	3.2
`assertCountEqual(a, b)`	*a* and *b* have the same elements in the same number, regardless of their order.	3.2

source: `https://docs.python.org/3/library/unittest.html`

Logging

- important: get insight into flow of program
- `print` statements: not always best choice
- see `https://realpython.com/python-logging/`
- Module: `logging`
- First example:

```
import logging

logging.warning('Watch out!')
logging.error('I told you so: ERROR')
logging.critical("Now it's getting serious!")
```

Logging – Which Way to Go?

Task you want to perform	The best tool for the task
Display console output for ordinary usage of a command line script or program	`print()`
Report events that occur during normal operation of a program (e.g. for status monitoring or fault investigation)	`logging.info()` (or `logging.debug()` for very detailed output for diagnostic purposes)
Issue a warning regarding a particular runtime event	`warnings.warn()` in library code if the issue is avoidable and the client application should be modified to eliminate the warning `logging.warning()` if there is nothing the client application can do about the situation, but the event should still be noted
Report an error regarding a particular runtime event	Raise an exception
Report suppression of an error without raising an exception (e.g. error handler in a long-running server process)	`logging.error()`, `logging.exception()` or `logging.critical()` as appropriate for the specific error and application domain

source: `https://docs.python.org/3/howto/logging.html`

Logging – Levels & Files

- Different log levels:

Level	When it's used
DEBUG	Detailed information, typically of interest only when diagnosing problems.
INFO	Confirmation that things are working as expected.
WARNING	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
ERROR	Due to a more serious problem, the software has not been able to perform some function.
CRITICAL	A serious error, indicating that the program itself may be unable to continue running.

source: `https://docs.python.org/3/howto/logging.html`

- Logging to file

```
import logging as lg
lg.basicConfig(filename='logTofile_ex.log',
               level=lg.DEBUG)
lg.debug('This␣message␣should␣go␣to␣the␣log␣file')
lg.info('So␣should␣this')
lg.warning('And␣this,␣too')
```

Anonymous Functions

- no name at construction
- keyword `lambda`
- can be inserted at arbitrary positions in code
- may get names assigned

```
>>> inc = lambda(i): i+1
>>> inc(3)
4
```

- come in handy for single-line usage

```
from numpy import *

fromfunction(lambda i,j: i+j, (3,4), dtype=int)
```

Anonymous Functions, Map & Filter

map

- `map`: applies a function to all elements of a list
- function: may be anonymous

```
>>> L = range(10)
>>> list(map(lambda x: x*x+1, L))
[1, 2, 5, 10, 17, 26, 37, 50, 65, 82]
```

filter

- `filter`: applies a function to all elements of a list
- elements returned: those for which the function returns `True`

```
>>> list(filter(lambda x: x%2==0, L))
[0, 2, 4, 6, 8]
```

Time

Module `time`

- Functions to access and display (the current) timestamp
- Without parameter: use current time

```
from time import *
time()                  # secs since 01/01/1970
ctime([secs])           # string (local time)
localtime([secs]) # struct (local time)
gmtime([secs])          # struct (GMT)
# Y, M, D, H, M, S, Wday, Yday, DST (1=y,0=n,-1=?)
sleep(secs)             # pause curr. process
```

- Measure time (up to 1/100s, system clock dependent)

```
t = time()
for i in range(10000):
    # do something
t = time()-t
print(t, "seconds")
```

Module `argparse`

- allows for comfortable parsing of command-line arguments to scripts
- We need to
  - create a parser
  - specify the arguments
  - trigger the parsing
- first example:

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("a") # positional argument
args = parser.parse_args()

print(args.a)
```

Module argparse – Adding Arguments

```
# positional argument:
parser.add_argument( 'name', options... )
# single-char flag:
parser.add_argument( '-x', options... )
# flag with long name:
parser.add_argument( '-l', '--long-name', options... )
```

Module `argparse` – Options

Different options available:

- `required` when adding an argument (default: `False`)

```
parser.add_argument('-o','--output', required=True)
```

- `action`
- `type`
- `default`

```
parser.add_argument('--rho', action='store',
                    type=float, default = 0.1,
                    help="Density␣of␣whatever")
```

- more details and usage: check `argparse1.py` and `argparse2.py` of course website

Jupyter Notebook

- available locally in *anaconda*
  - Start anaconda navigator
  - Select “Launch” for “jupyter”

  $\Rightarrow$ Window opens in browser
  - Select notebook file or create new
- examples on course website:
  - `08-plotting-new.ipynb`
  - `tips-notebook.ipynb`

Function Decorators

- Allow for customization of existing functions (e.g., from packages)
- No need to change original functions
- Further info:

```
http://thecodeship.com/patterns/
guide-to-python-function-decorators/
```

Advanced Plotting

- Access to axes etc.
- LaTex support for axes etc.
- examples of course website (zip file):
  - `08-plotting-new.ipynb`
  - `plotAppl3D.py`

Extending Python using C/C++

- Performance critical parts can be written in hardware-dependent code
- Use existing functionality written in other languages

$\Rightarrow$ Use as module

- We'll built an example, using C++ and swig
- Provides two functions, computing mod and factorial

Extending Python using C/C++ (2)

- Example:

```
/* example.cpp */
#include "example.hpp"

int mod(int x, int y) {
  return (x%y);
}
int fact(int n) {
  if (n <= 1) return 1;
  else return n*fact(n-1);
}
```

```
/* example.hpp */
// ...
int mod(int x, int y);
int fact(int n);
// ...
```

Extending Python using C/C++ (3)

- Additionally, we need to care about
  - Data conversion from C++ to Python
  - Writing both wrapper and interface files
- Fortunately, swig (Simplified Wrapper and Interface Generator) does everything for us
- We only need a .i file:

```
/* tools.i */
%module tools
%{
/* Put header files here ... */
#include "example.hpp"
%}
/* Parse the header file to generate wrappers */
%include "example.hpp"
```

Extending Python using C/C++ (4)

- Now, we've only got to run swig

```
swig -o tools_wrap.cpp -c++ -python tools.i
```

creating `tools.py` and `tools_wrap.cpp` ...

- ... and compile everything as a shared library

```
g++ -o tools_wrap.os -c -I/usr/include/python2.7 \
  tools_wrap.cpp -fPIC
g++ -o example.os -c example.cpp -fPIC
g++ -o _tools.so -shared example.os tools_wrap.os
```

- Then use in Python:

```
import tools
print(tools.mod(8, 6))
print(tools.fact(6))
```

Extending Python using C/C++ (5)

Comments

- Using global variables is a little more tricky
- Functions can be conveniently renamed in the .i file

```
%rename(__str__) DataVector::toString;
```

Faster Code

- Always good ;-)
- Never optimize before measuring!
  Policy:
  1. Get it right.
  2. Test it's right.
  3. Profile if slow.
  4. Optimise.
  5. Repeat from 2.
- Further info:

`https://wiki.python.org/moin/PythonSpeed/PerformanceTips`

More Python Fun :-)

Web server in 3 lines?

```
import SimpleHTTPServer, SocketServer
httpd = SocketServer.TCPServer(("", 8000), \
        SimpleHTTPServer.SimpleHTTPRequestHandler)
httpd.serve_forever()
```

- What could this be good for?
  - Want to share quickly some files with colleagues in the same network?

  $\Rightarrow$ Goto directory, start python, run three lines, tell them your IP and the port (here: 8000)
  - That's it!

```
import sys
sys.exit(0)
```