Script language: Python

Data and files

Cédric Saule

Technische Fakultät
Universität Bielefeld

13. Februar 2016

Python

–

User inputs, user outputs

Command line parameters, inputs and outputs of user data.

Command line parameters – `argparse` module

So far, command line parameters are directly known through the `sys.argv` process. `argparse` offers a very elegant object-oriented approach for processing.

- Auto-generated help for the user.
- Positional und optional parameters.
- Group of parameters.
- Type checking.
- Object oriented access.

Create a parser → Add arguments and groups → Parse → ggf. Error output after check → handle arguments.

Constructor `prsr = argparse.ArgumentParser()`

Argument If the arguments' name begins with one or more „-“, it is optional, otherwise it is positional.

```
prsr.add_argument(ANAME[,
 LONGANAME, help="...", type=T,
  action="..."])
```

Exclusive groups Exclusive parameters are grouped.

```
grp = prsr.add_mutually_exclusive_group()
grp.add_argument(...)
```

Parse `args = prsr.parse_args()`

argparse module – Example

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("square",
  help="display␣a␣square␣of␣a␣given␣number",
  type=int)
args = parser.parse_args()
print args.square**2
```

```
$ python prog.py 4
16
$ python prog.py four
usage: prog.py [-h] square
prog.py: error: argument square: invalid int value: 'four'
```

The object `args`

Effect of the `parse_args()` method call:

- Type checking for all parameters.
- Checks for the presence of positional parameters.
- Satisfiability of the grouped parameters.
- Creates the returned object, each argument is assigned to a field accessed by `args.NAME`.

argparse module – Example

```
import argparse

parser = argparse.ArgumentParser(description="calculate␣X^Y")
parser.add_argument("x", type=int, help="the␣base")
parser.add_argument("y", type=int, help="the␣exponent")

group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")

args = parser.parse_args()
answer = args.x**args.y

if args.quiet:
    print answer
elif args.verbose:
    print "{}␣to␣the␣power␣{}␣equals␣{}".format(args.x,
      args.y, answer)
else:
    print "{}^{}␣==␣{}".format(args.x, args.y, answer)
```

Exercises – `argparse` + `Fasta`

Extend your FASTA generation script in order to be able to deal with command line parameters.

These switches / arguments should be implemented:

- LENGTH Size of the Fasta-Sequence.
- NAME Name of the data set.
- -d, -r, -a Exclusionary group:
  - -d DNA-Alphabet (ATCG).
  - -r RNA-Alphabet (AUCG).
  - -a Amino acid alphabet (ACDEFGHIKLMNPQRSTVWY).

Interactive inputs – raw_input()

The method raw_input([prompt]) can be used to read inputs during the execution of a Python script.

```
> s = raw_input("Movie␣name:␣")
Movie name: Snakes on a Plane
> s
"Snakes␣on␣a␣Plane"
```

The method returns a prompt in STDOUT and waits for user input. The input is terminated by an "end of line" and delivers string (stripped).

Variables outputs

So far neglected: output of multiple variables as a string, or the concatenation of all these.

```
> z = 123
> print z+"␣is␣divisible␣by␣"+3+"."
Error type: cannot concatenate 'str' and
  'int' objects
```

Solution: `str()` and `repr()`

```
> print str(z)+"␣is␣divisible␣by␣"+str(3)+"."
123 is divisible by 3.
```

Variables outputs

str(<V>) Returned value: human readable string presentation.

repr(<V>) Returned value: machine readable string (Interpreter processed)

```
> str(s)
'Hello, world.'
> repr(s)
"'Hello, world.'"
> str(1.0/7.0)+" "+repr(1.0/7.0)
'0.142857142857 0.14285714285714285'
```

Variables outputs – Strings formating

By using `str.format()` prints the string (the format can be changed) `{}` are placeholders.

Automatic

```
> print '{}␣and␣{}'.format('spam', 'eggs', 42)
spam and eggs
```

Positional

```
> print '{0}␣and␣{1}'.format('spam', 'eggs', 42)
spam and eggs
> print '{1}␣and␣{2}'.format('spam', 'eggs', 42)
eggs and 42
```

No mix allowed!

Variables outputs – Strings formating

A third opportunity exists for the use of *keywords* which can be used as desired with any of the other methods.

```
> print 'This␣{food}␣is␣{adjective}.'.format(
  food='spam', adjective='absolutely␣horrible')
This spam is absolutely horrible.

> print 'The␣story␣of␣{0},␣{other}␣and␣{1}.'.
  format('Bill', 'Manfred', other='Georg')
The story of Bill, Georg, and Manfred.
```

Important : The *keywords* must stand near the „normal“ variables.

Variables outputs – Strings formating

Numbers accuracy can be formatted in the following way:
`{P:C[V.N|V]F}`:

- P Position.
- C Flags for signs.
- V Attributes to be placed to the left.
- : Introductory formatting characters.
- .N Decimal precision in floating point numbers.
- F Variable type.

```
> print '{:+08d}␣PI␣is␣~{:7.3f}.'.
    format(1, math.pi)
+0000001 PI is ~  3.142.
|       |         |      |
01234567          1234567
```

Details:
http://docs.python.org/2/library/stdtypes.html#string-formatting

Python

–

Reading and writing data.

Files, Streams

File access

The `open(<FILENAME>[, <MODE>])` method returns a `file` object, for reading and writing, as defined in the standard methods.

`FILENAME` gives the name of the file to be opened (default: relative path SPECIFIED). The standard mode is for reading, an alternative `MODE` (`string`) can be specified:

- r Read only – default
- r+ Reading and writing.
- w Write only.
- a Write append.

Data access – Reading

Three standard methods for reading a file:

read([size])	Entirely read, otherwise reads `size` Bytes. Returned type: `string`
readlines()	Entirely read. Returned type: `list` of `strings`
readline()	Reads a single line. Returned type: `string`

We can noticed that :

- Line endings are not removed.
- If no more row can be read, the returned value is always an empty string.
- Finally `close()` should *always* be called

Access files – reading

Here, we give a code example with loops and errors handling.

```
> for line in f:
>   print line,
This is the first line of the file.
Second line of the file

> with open('workfile', 'r') as f:
>   read_data = f.read()
> f.closed
True
```

Access files – writing

Two methods to write in a file:

write(<STRING>) `STRING` will be written in a file.

writelines(<SEQ<STRING> >) all the strings in the given sequence, will be written in the file.

We can noticed that :

- The " end of line " will not be automatically added.
- `writelines()` sequence of strings are expected as parameters.
- `write()`/`writelines()` only able to process strings!

Access files – writing

```
> f = open('the.question', 'w')
> f.readline()
''

> value = ('the␣answer', 42)
> s = str(value)
> f.write(s)

> f.seek(0)
> f.readline()
"('the␣answer',␣42)"
> f.readline()
''
> f.close()
```

Access files – STDIN

The access to the content of STDIN is performed in the same way as in a standard file object.

Here the sys module has to be imported, and then the object sys.stdin is at your disposal so that all read methods possess file.

```
> # count.py
> import sys
> data = sys.stdin.readlines()
> print "Counted", len(data), "lines."

$ cat count.py | ./count.py
Counted 3 lines.
```

Exercise – FASTA Im-/Export

Improve your FASTA program with the following functionalities:

1. Export data when a filename is specified as an additional parameter.
2. Read a file over STDIN, in which a new alphabet is set. A new character is on every line. The alphabet `{A, U, 1, .}` is defined by the following example file:

```
1|  A
2|  U
3|  1
4|  .
```

3. Use `sys.stdin.isatty()` to check STDIN.