Tutorium

Stefan Kurtz

Research Group for Genome Informatics
Center for Bioinformatics Hamburg
University of Hamburg

December 12, 2019

The versatility of for-loops (1/1)

```
for a in s:
  print('%{}'.format(a))

length_list = list()
for line in stream:
  length_list.append(len(line))

poem = 'Almost nothing was more'
char_list = list()
for cc in poem:
  if not (cc in char_list):
    char_list.append(cc)

for idx in range(25):
  print('%{}'.format(idx))

for idx, a in enumerate(s):
  print('%{}\t{}'.format(idx,a))
```

Nested for-loops (1/1)

```
for  i in range(1,n+1):
  for j in range(i+1,n+1):
    print('%{}\t{}'.format(i,j))
```

```
for  i in range(1,n+1):
  for j in range(0,i):
    print('%{}\t{}'.format(i,j))
```

Try/except, but only if necessary (1/1)

```
try:
  i = int(s)
except ValueError as err:
  sys.stderr.write('{}: {}\n'.format(sys.argv[0],err))
  exit(1)
```

```
try:
  arg = sys.argv[1]
except KeyError as err:
  sys.stderr.write('Usage: {} <argument>\n'.format(sys.argv[0]))
  exit(1)
```

```
if len(sys.argv) < 2:
  sys.stderr.write('Usage: {} <argument>\n'.format(sys.argv[0]))
  exit(1)
```

With or without Try/except when opening files (1/1)

```
with open(__file__) as stream:
  for line in stream:
    length_list.append(len(line))
```

```
Traceback (most recent call last):
  File "./notions.py", line 61, in <module>
    with open('xx') as stream:
FileNotFoundError: [Errno 2] No such file or directory: 'xx'
```

```
try:
  stream = open(__file__)
except IOError as err:
  sys.stderr.write('{}: {}\n'.format(sys.argv[0],err))
  exit(1)
```

```
./notions.py: [Errno 2] No such file or directory: 'xx'
```

Store lines only when necessary (1/1)

```
lines = stream.readlines()
for line in lines:
  length_list.append(len(line))

for line in stream:  # better, as more space efficient
  length_list.append(len(line))
```

Use join to combine lists of strings (1/1)

```
lss = ['a','b','c','d']
for i in range(len(lss)):
  s = '\t' if i < len(lss) - 1 else '\n'
  print('{}{}'.format(lss[i],s),end='')
print()


print('\t'.join(lss))
```

Splitting strings and sentences (1/1)

```
my_string = 'abcd'
print('%{}'.format(list(my_string)))

my_sentence = 'abracadabra, dreimal schwarzer kater'
print('%{}'.format(re.findall(r'\w+',my_sentence)))
print('%{}'.format(my_sentence.split()))
```

Slicing a string (1/1)

```
this_string = 'abcd'
for i in range(len(this_string)):
  print('%{} is suffix'.format(this_string[i:]))
  print('%{} is prefix'.format(this_string[:i+1]))

print('%{}'.format(this_string[-1]))

print('%{}'.format(this_string[::-1]))
```

Format gives you control over spaces (1/1)

```
f = 0.6
print('%f=',f)
print('%f={}'.format(f))
```

Providing eval with variable bindings (1/1)

- Material for 2nd Tutorium starts here
- you already know that the function `eval` can take as argument any string which is a correctly formed expression in Python
- it returns the value of the evaluated expression

```
result = eval('[1,2,3].append(4)')
print('%{}'.format(result))
```

- useful in contexts where a part of a Python program is dynamically created (e.g. a web-server))
- we can use variables in the evaluated expressions, if we provide bindings for the variables in form of a dictionary

```
'''
result = eval('[x,y,z].append(1)')  # variables x y z are not known
'''
result = eval('[x,y,z].append(4)',{'x' : 1, 'y' : 2, 'z' : 3})
print('%{}'.format(result))
```

Pitfalls when reading lines (1/1)

```
with open('tmpfile','w') as stream:
  for num in [1,15,100]:
    stream.write('{}\n'.format(num))

with open('tmpfile','r') as stream:
  for line in stream:
    print('%{}'.format(len(line)))
```

What does the +-operator do? (1/1)

```
with open('tmpfile','r') as stream:
  my_sum = None
  for num in stream:
    num = num.rstrip()
    if my_sum is None:
       my_sum = num
    else:
      my_sum += num
  print('%my_sum={}'.format(my_sum))
```

What does this program do? (1/1)

```
im_re = '(-?\s*\d+)\s*i'
for cs in ['7i-2','4-3i','9 i','-2',\
           '21-14i','-17 i+1']:
  m = re.search(r'{}'.format(im_re),cs)
  if not m:
    b = 0
    a = int(cs)
  else:
    b = int(m.group(1))
    if b < 0:
      b = '({})'.format(b)
    rest = re.sub(r'{}'.format(im_re),'',cs)
    if rest == '':
      a = 0
    else:
      a = int(rest)
  print('%{:10s} {}+{}i'.format(cs,a,b))
```

Representing a matrix (1/2)

0	0	0	0	0	0	0	0	0	0
0	0	0	4	0	0	0	0	0	0
0	0	0	0	7	0	0	0	0	0
5	0	0	0	0	0	0	0	0	4
0	0	0	0	0	0	0	0	8	0
7	0	0	0	0	0	7	0	9	0
0	0	0	0	0	6	0	0	0	0
0	0	4	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
3	2	0	0	0	0	7	1	0	0

- how can such a matrix be represented?
- if the matrix is sparse (i.e. most values are constant such a 0), one can write the values as a list:

```
values_list = [(9, 0, 3), (1, 3, 4), (9, 1, 2), (4, 8, 8), (3, 0, 5), (8, 1, 1), (5, 6, 7), (5, 8, 9),
 (3, 9, 4), (5, 0, 7), (7, 2, 4), (9, 6, 7), (2, 4, 7), (6, 5, 6), (9, 7, 1)]
```

Representing a matrix (2/2)

- how can we turn the list into an matrix data structure that can efficiently be updated and retrieved

```
sparse_matrix = dict()
for i,j,v in value_list:
  sparse_matrix[(i,j)] = v
```

- why can we use `(i,j)` as key for dictionary?
- why could we not use `[i,j]` as key for dictionary?

Different division operators (1/1)

- what is the difference between `/` and `//`

```
print("{}".format(5/2))
print("{}".format(5//2))
```

Looking at some example exercises (1/7)

- exercise: compute Quersumme for number given as string via `sys.argv`

```
#!/usr/bin/env python3
import sys, re

if len(sys.argv) != 2:
  sys.stderr.write('Usage: {} <integer>\n'
                   .format(sys.argv[0]))
  exit(1)

number_string = sys.argv[1].strip()
if not re.search('^[-+]?\d+$', number_string):
  sys.stderr.write('{}: argument "{}" is not an integer\n'
                   .format(sys.argv[0],sys.argv[1]))
  exit(1)

quersumme = 0
for cc in number_string:
  if cc != '-' and cc != '+':
    quersumme += ord(cc) - ord('0')
print("{}\t{}".format(number_string, quersumme))
```

Looking at some example exercises (2/7)

- exercise: convert data in format `DD.MM.YYYY` to number of day in year

```
for line in stream:
  line = line.rstrip()
  mo = re.search(r'(\d{2})\.(\d{2})\.(\d{4})',line)
  if not mo:
    sys.stderr.write('line {} has incorrect format\n'.format(line))
    exit(1)
  day = int(mo.group(1))
  month = int(mo.group(2))
  year = int(mo.group(3))
  dayofyear = 0
  for m in range(1,month):
    if m == 2:
      if year % 400 == 0 or (year % 4 == 0 and year % 100 != 0):
        daysinmonth = 29
      else:
        daysinmonth = 28
    elif m == 4 or m == 6 or m == 9 or m == 11:
      daysinmonth = 30
    else:
      daysinmonth = 31
    dayofyear += daysinmonth
  dayofyear += day
  print('{}\t{}'.format(line,dayofyear))
stream.close
```

Looking at some example exercises (3/7)

- here is a function based implemenation using a dictionary to store the number of days for each of the 12 month

```
def is_leap_year(year):
  return year % 400 == 0 or (year % 4 == 0 and year % 100 != 0)

def date2daynum(year,month,day):
  daysinmonth = {1: 31, 2: 28, 3: 31,  4: 30,  5: 31,  6:30,
                 7: 31, 8: 31, 9: 30, 10: 31, 11: 30, 12:31}
  dayofyear = 0
  assert month <= 12
  for m in range(1,month):
    dayofyear += daysinmonth[m]
    if m == 2 and is_leap_year(year):
      dayofyear += 1
  dayofyear += day
  return dayofyear
```

- how could we use a list for the days in each month?

Looking at some example exercises (4/7)

– here is the corresponding main program (`args.inputfile` was set by an argument parser)

```
try:
  stream = open(args.inputfile,'r')
except IOError as err:
  sys.stderr.write('{}: {}\n'.format(sys.argv[0],err))
  exit(1)

for line in stream:
  line = line.rstrip()
  mo = re.search(r'(\d{2})\.(\d{2})\.(\d{4})',line)
  if not mo:
    sys.stderr.write('{}: line {} has incorrect format\n'
                     .format(sys.argv[0],line))
    exit(1)
  day = int(mo.group(1))
  month = int(mo.group(2))
  year = int(mo.group(3))
  dayofyear = date2daynum(year,month,day)
  print('{}\t{}'.format(line,dayofyear))
stream.close
```

Looking at some example exercises (5/7)

– here is a class for dates

```
class Date:
  daysinmonth = {1: 31, 2: 28, 3: 31,  4: 30,  5: 31,  6:30,
                 7: 31, 8: 31, 9: 30, 10: 31, 11: 30, 12:31}
  def __init__(self,dstring):
    mo = re.search(r'(\d{2})\.(\d{2})\.(\d{4})',dstring)
    if mo:
      self._day = int(mo.group(1))
      self._month = int(mo.group(2))
      self._year = int(mo.group(3))
    else:
      mo = re.search(r'(\d{4})-(\d{2})-(\d{2})',dstring)
      if mo:
        self._year = int(mo.group(1))
        self._month = int(mo.group(2))
        self._day = int(mo.group(3))
      else:
        raise Exception('"{}" is not a valid date'.format(dstring))
  def date2number(self):
    dayofyear = 0
    assert self._month <= 12
    for m in range(1,self._month):
      dayofyear += Date.daysinmonth[m]
      if m == 2 and is_leap_year(self._year):
        dayofyear += 1
    dayofyear += self._day
    return dayofyear
  def __str__(self):
    return '{:02d}.{:02d}.{}'.format(self._day,self._month,self._year)
```

Looking at some example exercises (6/7)

– here is the corresponding main program (`args.inputfile` and `args.day2number` were set by an argument parser)

```
try:
  stream = open(args.inputfile,'r')
except IOError as err:
  sys.stderr.write('{}: {}\n'.format(sys.argv[0],err))
  exit(1)

for line in stream:
  line = line.rstrip()
  try:
    dt = Date(line)
  except Exception as err:
    sys.stderr.write('{}: {}\n'.format(sys.argv[0],err))
    exit(1)
  values = [str(dt)]
  if args.day2number:
    values.append(str(dt.date2number()))
  print('\t'.join(values))
stream.close
```

Looking at some example exercises (7/7)

– exercise: fold text into lines of given maximum width

```
for line in stream:
  out_list = list()
  sum_of_outlist = 0
  for string in re.findall(r'\S+',line):
    if sum_of_outlist + len(out_list) - 1 + \
       1 + len(string) > linewidth:
      print(' '.join(out_list))
      out_list = list()
      sum_of_outlist = 0
    out_list.append(string)
    sum_of_outlist += len(string)
  print(' '.join(out_list))
```

Difference and commonalities of ... (1/1)

1. the following methods:
   - `re.search`,
   - `re.sub`,
   - `re.findall`.
   - `s.translate`,
2. the following classes:
   - strings: `str()`
   - lists: `list()`
   - dictionaries: `list()`
   - sets: `set()`

Using pysearch.py

- pysearch.py -d dict
- pysearch.py dict
- pysearch.py -d list
- pysearch.py -d except
- pysearch.py -d listitems
- pysearch.py -d regexp
- pysearch.py split
- pysearch.py -d find

Assertions (1/2)

- use `assert` to
  - verify conditions that are supposed to hold before some statements
  - document your own requirements
- a statement `assert condition` is basically the same (except for the error message output) as:

```
if not condition:
  sys.stderr.write('{} fails'.format(str(condition)))
  exit(1)
```

- assertions allow to detect programming errors
- assertions can be combined with user defined messages to be output if the assertion fails

Assertions (2/2)

- to ignore assertions, use `python3 -O` or `export PYTHONOPTIMIZE=TRUE`

```
$ echo "assert False" | python3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
$ echo "assert False" | python3 -O
```

- for errors due to user input (like files that cannot be opened), use corresponding tests or exception handling, but not assertions

More topics

- how to debug programming code
- piping into python3
- using python3 in interactive mode

```
$ echo "print(sum(list(range(1,10))))" | python3
45
```