Programming and Algorithms

Python cheatsheet

Variables and data types

- Variables may contain different kinds of information, and be used in different ways depending on their content
  - Numbers:

```
x = 2
y = x + 3
if y>0 :  ...     # y is 5, so y>0 would be True
```

  - Strings or characters

```
x = 'a'
y = 'hello'
if x != 'b' : ...     # x contains 'a', so it would be True
if x in y : ...       # False, since 'a' is not contained in 'hello'
```

  - Boolean values
    - ok = **True**

```
if ok :  ...              # True, since ok is True
if not ok :  ...          # False, since ok is True
```

Variables and data types

- Variables may also contain groups or collections of things
  - Tuples. Pack several values toghether in an inmutable object

```
x = (1, 'a', 3, 'bye')
a, b, c, d = x
# a==1, b=='a', c==3, d=='bye'
```

  - Lists. Pack several values toghether in a mutable object

```
x = [1, 'a', 3, 'bye']
x.append(8)     # now x is [1, 'a', 3, 'bye', 8]
x[2] = 5        # now x is [1, 'a', 5, 'bye', 8]
```

  - Dictionaries. Pack several values toghether in a mutable object with key-value pairs

```
x = {'hello': 3, 'bye': 11, 'you': 7}
if 'bye' in x  : ...     # True, since x contains key 'bye'
if x['hello'] == 4 : ...   # False, since value  for key 'hello' is 3
```

Functions

- Functions expect parameters and *return* the result of some computation on them. They need to be called from a main program or from another function.

```
def myfun(x) :      ## x is expected to be an integer
    y = x + 2
    if y%2 == 0:
        return y//2
    else :
        return y*3      ## The result of the function will be int


def myfun(x,y) :      ## x is expected to be a char and y a string
    if x in y:
    return True
    else :
    return (x=='a' or x=='u')    ## result will be a boolean
```

Elements in tuples, strings, lists, or dictionaries

- We can access individual elements inside structured variables
  - x = ‘hello’
    if x[0] == ‘h’ : .... # True, first letter in x is ‘h’
    if x[-1] == ‘o’ : .... # True, last letter in x is ‘h’
    i = 2
    if x[i] == ‘l’ : ... # True, third letter in x is ‘l’
  - t = (1, ‘a’, 3, ‘bye’)
    if t[2] + t[0] == 4 : ... # True, 3+1 is 4
  - p = [1, 6, 2, 12, 9]
    k = 1
    if p[3] + p[k] == 18 : ... # True, p[3]+p[1] = 12+6 = 18
  - d = {‘hello’: 3, ‘bye’: 4, ‘you’: 1}
    if d[‘hello’] == d[‘bye’] : ... # False, 3!=4
    d[‘hello’] += 1
    if d[‘hello’] == d[‘bye’] : ... # True, both are 4

Loops

- We can loop over all elements in a tuple or string

```
w = ‘hello’
for c in w :
    do_something    # will be repeated with c having the
                    # value of  each character in w


t = (1, ‘a’, 2, ‘hello’)
for e in t :
    do_something    # will be repeated with e having the
                    # value of  each element in t
```

Loops

- We can also loop over all elements in list:

```
for k in [1, 4, 12, 6, 10, 23]:
    do_something    # will be repeated with k having each
                    # of the listed values

for k in [‘hello’, ‘bye’, ‘today’]:
    do_something
```

Loops

- We can also express lists of numbers easily:

```
for k in range(0,10) :  # range(0,10) is the same
    do_something        # than [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

for k in range(0,10000) :  # much shorter this way
    do_something           # than writting them all

for k in range(a,b) :    # different lists depending on a and b
    do_something
```

Loops

- **in** can be used also with **if**

```
if k in [1, 4, 12, 6, 10, 23]:
    do_something    # will be executed if k is one of these
```

- **if** k **in** [‘hello’, ‘bye’, ‘today’]:

```
    do_something
```

- **if** k **in** [‘a’,‘e’,‘i’,‘o’,‘u’]:

```
    do_something
if k in ‘aeiou’:
    do_something
```

Loops

- We can also repeat arbitrary instructions as long as needed

```
s = 0
while n>0 :          # No explicit list of elements.
    s = s + n%10     # But if e.g. n == 3512, its value will
    n = n//10        # change at each iteration, and n will be
                     # sucessively: 3512, 351, 35, 3, and 0
```

Reading data (with sys.stdin)

- We can read data from standard input (usually keyboard, but could be also a file)

```
import sys

line = sys.stdin.readline()   # will read a whole line as a string
# e.g. line == 'hello bye today\n'
words = line.split()   # strings can be splitted into tokens
# words == ['hello', 'bye', 'today']

line = sys.stdin.readline()   # will read a whole line as a string
# e.g. line == '23  5  141  12\n'
words = line.split()   # strings can be splitted into tokens
# words == ['23', '5', '141', '12']
numbers = []
for w  in words :                    # if strings are numbers,
    numbers.append(int(w))    # they can be converted to integers
# numbers == [23, 5, 141, 12]
```

Reading data (with easyinput)

- We can also use *easyinput* module to read data from stdin

```
from easyinput import read, read_line

line = read_line()   # same as sys.stdin.readline()

word = read(str)     # will read a string until the first whitespace or enter
num = read(int)      # will read a number
```

- With *easyinput* there is no need to split string or to convert strings to integers.
  We can read what we need when we need.
- Warning: Easyinput *read* treats newlines the same than whitespaces, so there is no sense of «line».
  Use read_line (or sys.input.readlines) if you need to process lines.

Frequent loop patterns

- Read a known number of elements

```
from easyinput import read

n = read(int)      # number of things to be read
for _ in range(n):   # repeat n times
    x = read(int)    # read number (or str, or whatever is needed)
    process(x)       # do something with x (count, add, store...)
```

Frequent loop patterns

- Read an unknown number of elements, until the end of file

```
from easyinput import read

x = read(int)                # read first element (int, string, line, etc...)
while x is not None :        # x will be None when read find end-of-file
    process(x)               # do something with x (count, add, store...)
    x = read(int)            # read next element
```

Input example:

```
1 5 7 12 2 3
5 8 9
4 22 44 1
```

Frequent loop patterns

- Read an unknown number of elements, until a special one is found

```
from easyinput import read

x = read(int)          # read first element (int, string, line, etc...)
while x != 0 :         # stop when a special value is read (zero, -1, 'end'...)
    process(x)         # do something with x (count, add, store...)
    x = read(int)      # read next element
```

Input example:

```
1  5  7  12  2  3
   5  8  9
4  22  44  1 0
```

Frequent loop patterns

- Read a known number of blocks, each with a known number of elements

  from easyinput import read

  - ```
    nb = read(int)          # read number of blocks
    for b in range(nb):
        ne = read(int)      # read number of elements in this block
        for e in range(ne) :
            x = read(int)       # read element e in block b
            process(x)
    ```

  Input example:

  ```
  3
  5   7  12  2  3  1
  2   8  9
  4   22  0  1 8
  ```
  ```

Frequent loop patterns

- Read an unknown number of blocks (until no more data), each with a known number of elements

```
from easyinput import read

ne = read(int)        # read number of elements of first block
while ne is not None:
    for e in range(ne) :
        x = read(int)   # read element e in block b
        process(x)

    ne = read(int)    # read number of elements in next block
```

Input example:

5 7 12 2 3 1
2 8 9
4 22 0 1 8

Frequent loop patterns

- Read an unknown number of blocks (until special value), each with a known number of elements

```
from easyinput import read

ne = read(int)        # read number of elements of first block
while ne != -1 :
    for e in range(ne) :
        x = read(int)   # read element e in block b
        process(x)
    ne = read(int)    # read number of elements in next block
```

Input example:

```
5   7 12 2 3 1
2   8 9
4   22 0 1 8
-1
```

Frequent loop patterns

- Read a known number of blocks, each with an unknown number of elements, a special element marks the end of each block

```
from easyinput import read

nb = read(int)          # read number of blocks
for b in range(nb):
    x = read(int)       # read first element in this block
    while x != 0 :
        process(x)
        x = read(int)       # read next element in block b
```

Input example:

```
3
 7  12  2  3  1 0
 8  9 0
 22  4  1 8 0
```

Frequent loop patterns

- Read an unknown number of blocks (until no more data), each with an unknown number of elements, a special element marks the end of each block

```
from easyinput import read

x = read(int)        # read first element of first block
while x is not None :
    while x != 0 :
        process(x)
        x = read(int)   # read next element in current block


    x = read(int)    # read first element of next block
```

Input example:

```
7  12  2  3  1 0
8  9 0
22  4 1 8 0
```

Frequent loop patterns

- Read an unknown number of blocks (until special value), each with an unknown number of elements, a special element marks the end of each block

```
from easyinput import read

x = read(int)        # read first element of first block
while x != -1 :
    while x != 0 :
        process(x)
        x = read(int)   # read next element in current block


    x = read(int)    # read first element of next block
```

Input example:

```
7 12 2 3 1 0
8 9 0
22 4 1 8 0
-1
```