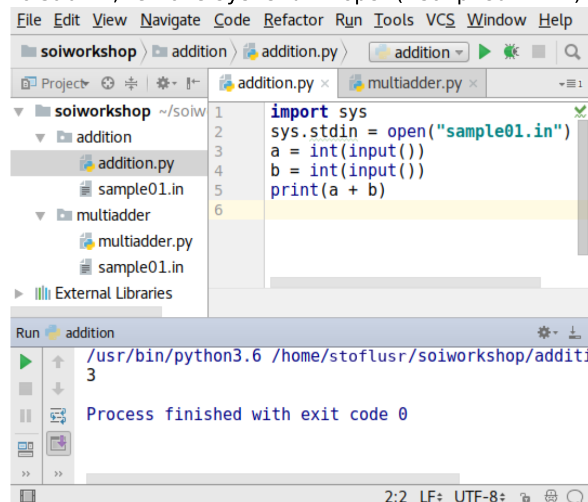
Python Cheat Sheet – SOI Workshops 2017

Python Docs

https://docs.python.org/

Recommended PyCharm Setup

To submit, remove `sys.stdin=open("sample01.in")`.



I/O

```
string = input()         # read line as string
number = int(input())    # read line as integer
a, b = map(int, input().split())  # read two integers
numbers = list(map(int, input().split()))  # read list
print(" ".join(map(str, numbers)))  # print list

print(value)            # print string/int/... on a line
print(value, end="")    # don't start a new line
print(a, b)             # print two values sep. by space
print("impossible")     # print a string literal
print(f"Case #{testcase}: {solution}")  # formatting
#      ^-- the f is important and stands for formatted
```

Files

```
import sys
sys.stdin = open("sample01.in")
sys.stdout = open("sample01.out", "w")
```

Remove those lines before submitting at grader.soi.ch!

Operators

- Comparison: ==, !=, <, <=, >, >=
- Logical: and, or, not
- Arithmetic: +, -, *, ** (pow: 3**4=81), / (float division: 7/3=2.33), // (integer div.: 7//3=2), % (modulo: 7%3=1)
- Bitwise: &, |, ^, ~ (two's complement), <<, >>

Assignment statement: `a += 1` (works for most operators)

Loops

for loops iterate over a range:

```
for i in range(5):  # for loop
    print(i)  # prints 0, 1, 2, 3, 4
for i in range(2, 14, 4):  # start, stop, step
    print(i)  # prints 2, 6, 10
for _ in range(7):  # use _ if you don't need the idx
    do_work()        # will be executed 7 times
```

Tuples

```
tup = 3, 4, 5  # create tuple of two elements
a, b, c = tup  # extract tuple: a=3, b=4, c=5
_, _, c = tup  # don't care: use _
```

Lists (can be used as stacks)

Basics:

```
numbers = []  # create new list
numbers = [3, 1, 4, 1, 5, 9]  # prefilled
numbers = [0 for _ in range(n)]  # length n, all 0
```

Iteration:

```
for x in numbers:  # iterate over list
    print(x)
for i, x in enumerate(numbers):  # index and element
    print(f"Value {x} is at index {i}.")
for i in range(len(numbers)):  # only index
    number[i] = 4  # change i-th element
```

Operations:

```
numbers.append(3) # O(1) append at the end
x = numbers.pop() # O(1) remove last element
numbers = [1] + numbers  # O(len(numbers))
numbers.index(3)  # O(n) index of value 3
```

Indexing:

- `a[i]`: i-th element of a: 0=first, `len(a)-1`=last
- `a[-1]`: last element ("`-2`" is second last, etc.)

Slices:

- `a[3:6]`: elements at indices 3, 4, 5
- `a[:3]`: elements at indices 0, 1, 2
- `a[3:]`: elements at indices 3, 4, ..., `len(a)-1`
- `a[:]`: copy of whole list

List comprehensions:

```
numbers = [3, 1, 4, 1, 5, 9]      # create new list
doubled = [2*x for x in numbers] # list comprehension
```

List functions:

```
sum(numbers)  # sum of all elements
max(numbers)  # maximum of all elements
min(numbers)  # minimum of all elements
max(numbers, key=f)  # compare with f(x)<f(y)
sorted(numbers)  # increasingly sorted copy
sorted(numbers, reverse=True)  # decreasing
sorted(numbers, key=f)  # compare with f(x)<f(y)
list(reversed(numbers))  # reverse list
```

Deques (can be used as queues)

```
from collections import deque
q = deque()      # empty
q = deque([2])   # prefilled
q.append(4)      # O(1) append at the end
q.appendleft(1)  # O(1) append at the front
x = q.pop()      # O(1) remove last element
y = q.popleft()  # O(1) remove first element
```

Min-Heaps/Priority Queues

Datastructure with fast insert and fast access to the minimum. (Trick: to sort it in reserve, insert -x.)

```
from heapq import *
pq = [];            # O(1)     declare a new heap
heappush(pq, 1);    # O(log n) insert elements
print(heappop(pq)) # O(log n) get and remove min.
```

Dictionaries

Stores key-value pairs with fast key lookup. Keys are unique.

```
m = {}  # create an empty dict
m = {"Turing": 1954: "Newton": 1727}  # prefilled
m = {x: 2**x for x in range(10)}      # list compr.
m["Einstein"] = 1955    # O(1) insert or modify
print("Einstein" in m) # O(1) check if key exists
del m["Einstein"]       # O(1) remove item
for key, value in m.items():  # O(len(m))) iterate
    print(f"key: {key} -> {value}")  # not sorted
```

Sets

Elements are unique. Like a dict, but only stores keys.

```
s = set() # create an empty set
s = {1, 2} # prefilled
s = set(values) # convert list to set
# WARNING: s = {} creates a dict, not a set
s.add(1)        # O(1) add element to set
print(1 in s)   # O(1) check if element exists
s.remove(2)     # O(1) remove element
for element in s: # O(len(s)) iterate
    print(s)    # careful: not sorted
```

Recursion Limit

Recursive functions may produce a "`RecursionError: maximum recursion depth exceeded`".

```
import sys
sys.setrecursionlimit(10**9)
```

Assertions

Assertions are checks that crash your program if they fail. Document what properties should hold using assert.

```
assert 0 <= i < len(a), "index valid"
assert len(a) > 0, "list not empty"
assert x <= 1000  # assertion without comment
```