Scripting Languages

Python basics

- lists & tuples
- ranges
- for-in loop
- functions
- exercise

Data type → `list`

Ordered collection of items

- items can be referenced by [index] (starting from 0)
- items can be of any type (including other lists)
- can contain items of various types
- can be modified (grow, shrink , item changes)
- can be joined, sliced, compared

```
x = [item, item, item, …]
```

```
>>> a = [1, 2, 3]
>>> b = ['apple','orange']
>>> c = [[3,'red'],[5,'blue']]
>>> print(a[0], b[1], c[1], c[1][0])
1 orange [5, 'blue'] 5
```

`list` cont.

Some operation on lists:

- `x[index] = newitem` → change existing item
- `x.insert(index, item)` → insert item at index position
- `x.append(item)` →add item at the end
- `del x[index]` → remove item at index position
- `x.remove(item)` → remove item (first occurrence)
- `len(x)` → return list length
- `item in x` → reports existence of item in list (True of False)

```
>>> a = [1, 2, 3]
>>> a.append(5)
>>> a.insert(3,4)
>>> print(a)
[1, 2, 3, 4, 5]
```

```
>>> del a[1]
>>> a.remove(4)
>>> 1 in a
True
>>> print(a)
[1, 3, 5]
```

Data type → `tuple`

Tuple → a fixed list

- tuples can story any types of objects
- access to elements is identical as to lists
- no modifications are allowed

  `x = (item, item, item, …)`

```
>>> a = (1, [2])
>>> print(a[0])
1
>>> a[0]=0
TypeError: 'tuple' does not support item assignment
>>> a[1][0]=0
>>> print(a)
(1, [0])
>>> a[1].clear()
>>> print(a)
(1, [])
```

Sequence generator → `range()`

Integers with arithmetic progression:

`range(stop)`

- range(3) → 0, 1, 2
- range(10) → 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

`range(start, stop [,step])`

- range(1, 3) → 1, 2
- range(-1, 2) → -1, 0, 1
- range(1, 11, 2) → 1, 3, 5, 7, 9
- range(0, -10, -1) → 0, -1, -2, -3, -4, -5, -6, -7, -8, -9

Making lists from ranges:

- `list( range(start, stop [,step]) )`

```
>>> a = range(10); print(a)
range(0, 10)
>>> b = list(a); print(b)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`list` vs `range`

- Lists are data structures
  - elements occupy space in memory
- Ranges are iterator objects
  - elements are generated on request
- Lists and ranges support:
  - in, len(), min(), max(), sum(), index(), count()
  - concatenation of ranges is not possible

```
>>> a = range(10)
>>> b = list(a)
>>> print(len(a), len(b))
10 10
```

Definite loop → `for-in`

Definite loop

- repeat the body defined number of times
- iterate over elements of any sequence:
  - range, list or tuple (also non-numerical)
- `break` and `continue` are applicable

```
for item in squence:
    instr
```

```
for x in [4, 2,-1]:
    print(x**2)
```

```
I=('pretty','smart','modest')

for s in I:
    print("I'm " + s)
```

Definite loop with list vs range

Iterating over progressive integer sequences is more efficient with `range()`

```
n = 10**10

for i in range(n):
    a += i
```

MemoryError

```
n = 10**10

for i in list(range(n)):
    a += i
```

- Lists allow for iteration over any sequence of objects

```
I=('pretty','smart','modest')

for s in I:
    print("I'm " + s)
```

```
for s in "Hello":
    print(s)
```

Functions

Function – sequence of instructions executed with a single instruction (function call)

- function can take parameters and return value(s)
- names created inside function are local
- names existing in outer scope are accessible for reading

```
def function( parameters ):
    instructions
    return [value(s)]
```

```
def fib(n):
    if n<=0: return 0
    a, b = 0, 1
    for i in range(n-1):
        a, b = b, a+b
    return b
...
print(fib(3))
```

```
def fibseq(n):
    seq = []
    for i in range(n):
        seq.append(fib(i))
    return seq
...
print(fibseq(10)) # list
```

This is implementation of Fibonacci sequence has poor performance and is here just for demonstration

Function documentation

The programmer should describe:

- functionality, parameters and return value
- used for built-in documentation system

```
def fib(n):
    """ Fibonacci number """
    if n<=0: return 0
    a, b = 0, 1
    for i in range(n-1):
        a, b = b, a+b
    return b

...
print(fib(3))
```

```
def fibseq(n):
    """
    Returns the sequence
    of n-Fibonacci
    numbers
    """
    seq = []
    for i in range(n):
        seq.append(fib(i))
    return seq

print(fibseq(10))
```

Exercise

- Problem: sequence of Hailstone Numbers
  - $c_{n+1} = ½·c_n$ if c_n is even
  - $c_{n+1} = 3·c_n+1$ if c_n is odd
  - stop when reaching 1
    - example: c=3 → 10, 5, 16, 8, 4, 2, 1

(*polish: liczby gradowe*)

```
def next_hsn(c):
    if c % 2 == 0:
        return c//2
    else:
        return 3*c+1
```

```
def hsn(c):
    sequence = [c]
    while c>1:
        c = next_hsn(c)
        sequence.append(c)
    return sequence
```

hsn(0) → [0]
hsn(1) → [1]
hsn(2) → [2, 1]
...

```
print(hsn(27))
```

Exercise

For all HS sequences starting form 1 up to 100 (optionally 1000):

1) find the length of longest sequence
2) find the start value of the longest sequence
3) find the highest value ever reached (with start and length)
4) find the sequence with highest sum of elements

Optionally:

5) find the most common sequence length
6) find the longest common subsequence