

Objects and Lists

1DV501 - Introduction to Programming

Jonas Lundberg, office B3024

`Jonas.Lundberg@lnu.se`

The slides are available in Moodle

September 19, 2020

The Python Test

The first Python Test takes place on Friday, October 23.

- ▶ A 2 hour test where you will handle 2-3 programming exercises
- ▶ It will be based on the Python material covered in Assignments 1 and 2.
- ▶ You must be able to handle all exercises to pass the test.
- ▶ Allowed help:
 - ▶ Your own laptop and your favorite IDE (e.g. Visual Studio Code)
 - ▶ Internet access to the Python Language Reference (<https://docs.python.org/3>)
 - ▶ You will not be given access to any lecture slides, your own assignment solutions, or any other Python resource.
- ▶ You will be monitored the whole time.
- ▶ A 2nd and 3rd attempt will be given in November and December
- ▶ Registration deadline: October 16 (at 23.55).
Registration is **mandatory** and will start in a few days in Moodle.

The Python Test - Distance vs Campus

- ▶ For students staying in Sweden
 - ▶ The test will take place at Campus Växjö or at Campus Kalmar.
 - ▶ We will not allow any student living in Sweden to take the test remotely.
 - ▶ Exact time and place for the test will be presented later on.

- ▶ For students staying abroad
 - ▶ Will be given an opportunity to take the Python Test remotely.
 - ▶ The test will be monitored using Zoom. You will be asked to setup a webcam (or mobile phone) in such a way that a video of you and your computer is in clear view during the test.
 - ▶ More instructions related to the distance version of the Python Test will be presented later on.

Today ...

- ▶ Objects and Classes
- ▶ String Objects
- ▶ Fraction Objects
- ▶ Lists (introduction)
- ▶ List Methods
- ▶ List Slicing
- ▶ If time permits
 - ▶ List Comprehensions
 - ▶ Multidimensional Lists
- ▶ Programming example

Reading instructions: Sections 9.1, 9.2, 9.4, 10.1-10.10, 10.13-10.15
(in textbook by Halterman)

Classes and Objects

- ▶ Primitive types (e.g. `int`) have simple values (e.g. 237) and operations (e.g. `+`, `-`, `*`, `/`).
- ▶ Entities like for example a bank account have more complex values
⇒ they require a mixture of multiple values to be correctly described
- ▶ A **class** is a definition of a more complex type
- ▶ Values of a class are called **objects** (or **instances** of a class)

<code>class bank_account</code>	Object 1	Object 2	Object 3
Owner:	Jonas	Henrik	Nils
No:	4758-8696	3246-9744	5432-2347
Balance:	34.345kr	8.456kr	97.654kr

- ▶ Classes (e.g. `bank_account`) have more complex values (e.g. Jonas, 4758-8696, 34.345kr)
- ▶ The current values associated with an objects (e.g. Jonas, 4758-8696, 34.345kr) is called the *object state*.

Methods

- ▶ Types like `int` have simple values like 237 and operations like `*`
- ▶ Classes have complex values and a set of operators called **methods**
- ▶ The class `string` has for example a method called `upper()`

```
s = "Hello"           # "Hello" is an object of type/class string

s = s.upper()         # Apply method on string object "Hello"
print(s)              # Output: HELLO
```

- ▶ A class defines properties of a given type of objects
- ▶ A class definition (often a separate file) is a bit of code defining:
 - ▶ Attributes: The data we associate with the class
(for example owner, account number, and saldo for a bank account)
 - ▶ Methods: Operations we can do on an object
(for example `update_balance` on `bank_account` object)

Method Calls

- ▶ Classes come with a specific set of operators called **methods**
- ▶ The methods of class A can only be applied on objects of class A
- ▶ Methods are called (applied) on variables referencing an object

```
s = "Hello"           # "hello" is an object of class string
print( s.upper() )    # Output: HELLO
```

- ▶ General pattern for a method call



- ▶ In this lecture we look at a few common classes from the Python library
- ▶ We will *not* create our own classes

The string class `str`

- ▶ All strings are objects of a predefined class `str`

```
print( type("Hello") )      # Output: <class 'str'>
```

- ▶ We create new string objects using double "Hi" or single quotes 'Hi'
- ▶ The string class `str` has many methods

```
s = "Hello"

print( s.upper() )          # Output: HELLO
print( s.count("l") )       # Output: 2
print( s.find("lo") )       # Output: 3
print( s.endswith("xxx") )  # Output: False
print( s.isalpha() )        # Output: True
```

- ▶ `s.count("l")` $\Rightarrow$ number of "l" in string `s`
- ▶ `s.find("lo")` $\Rightarrow$ first position of "lo" in string `s`
- ▶ `s.endswith("xxx")` $\Rightarrow$ True if string `s` ends with "xxx"
- ▶ `s.isalpha()` $\Rightarrow$ True if string `s` only contains letters

The Python Standard Library

- ▶ The string class `str` comes with many methods
- ▶ It is hard to remember all details about all methods
- ▶ The official documentation for the string class is:

`https://docs.python.org/3/library/`

- ▶ The documentation is called the **Python Standard Library**
- ▶ The website documents all Python's built-in types, classes and functions
- ▶ Hard reading since designed for professionals
- ▶ **The `docs.python.org/3/` documentation (and your IDE) will be your only help at the Python Test ⇒ Get familiar with it!**

Methods vs Built-in Function

- ▶ Many built-in functions in Python can also be applied on strings

```
s = "abcABC"

print( len(s) )           # Output: 6
print( min(s) )           # Output: A
print( max(s) )           # Output: c
print( min("aA1"))        # Output: 1
print( min("aA 1{"))      # Output: " " (whitespace)
print( max("aA 1{"))      # Output: {
```

- ▶ `min(s)` $\Rightarrow$ first character in alphabetical order
- ▶ `max(s)` $\Rightarrow$ last character in alphabetical order
- ▶ alphabetical order: First digits, then upper case, then lower case, other character are sorted based on their ASCII number (I think)
- ▶ Notice also how built-in functions are applied (e.g. `len(s)`) compared to how methods are applied (e.g. `s.upper()`)

The class Fraction

The module fractions contains a class Fraction

```
from fractions import Fraction

f1 = Fraction(1,2)           # Create Fraction object 1/2
f2 = Fraction(1,3)
fsum = f1+f2                 # Store 1/2 + 1/3 in variable fsum

print(f1, type(f1))         # Output: 1/2 <class 'fractions.Fraction'>
print(fsum)                 # Output: 5/6
print(fsum.numerator)       # Output: 5
print(fsum.denominator)     # Output: 6
```

- ▶ We create a Fraction object $1/2$ by calling a method `Fraction(1,2)`
Methods used to create new objects are called **constructors**
- ▶ Creating a new object of class A using a constructor named A, is the standard approach
- ▶ `fsum.numerator` is not a method call, we are accessing the attribute called `numerator` $\Rightarrow$ the data values representing the object state

Simple Fraction example

```
from fractions import Fraction

f = 0
for n in range(2,11):
    f = f + Fraction(1,n)  # 1/2 + 1/3 + 1/4 + ... + 1/10
print(f, float(f))        # Output: 4861/2520 1.928968253968254
```

- ▶ We introduce class `Fraction` just to show how a typical class is used
- ▶ Objects in a typical class are created using constructors
- ▶ String objects created using `" "` or `' '` is an exception
- ▶ The string object creation (and lists and tuples objects) is simplified since their creation is very common

Data Structures – Introduction

- ▶ We often need to handle large sets of data
- ▶ A *data structure* is a model for storing/handling such data sets
- ▶ Scenarios where data structures are needed
 1. Students in a course
 2. Measurements from an experiment
 3. Queue to get an apartment at our campus
 4. Telephone numbers in Stockholm
- ▶ Different scenarios require different data structure properties
 - ▶ Data should be ordered
 - ▶ Not the same element twice
 - ▶ Important that look-up is fast
 - ▶ In general: Important that operations X,Y,Z are fast
- ▶ Selecting data structure is a design decision $\Rightarrow$ might affect performance, modifiability, and program comprehension.
- ▶ **Today:** Lists, later on tuples, sets, and dictionaries

Introducing lists

```
l = [1,2,3,4,5]           # A list containing 1,2,3,4,5

print( l, type(l) )       # Output: [1, 2, 3, 4, 5] <class 'list'>
print( l[0], type(l[0]) ) # Output: 1 <class 'int'>
```

- ▶ A list like `[1,2,3,4,5]` is an object of class **list**
- ▶ We create lists using enclosing square brackets
- ▶ They represent a sequence of data, each value is called an **element**
- ▶ We can access individual element using square brackets like `l[0]`
- ▶ The first position is 0 $\Rightarrow$ `l[0]` is the first element
- ▶ `[1,2,3,4,5]` is an integer list, but we can create lists of any type (or with mixed types)
- ▶ `list` is a built-in type $\Rightarrow$ no need for any import statement

Manipulating lists

```
lst = [1,2,3,4,5]

lst[2] = 99                # Replace element at position 2
print(lst)                # Output: [1, 2, 99, 4, 5]

# Iterate over all list indices
for i in range( len(lst) ):
    print( lst[i], end=" ")    # Output: 1 2 99 4 5
print()

# Iterate over all list elements
for n in lst:
    print( n, end=" ")        # Output: 1 2 99 4 5
print()
```

- ▶ We can replace a list element using `lst[2] = 99`
- ▶ Iteration using indices: `for i in range( len(lst) ):`
- ▶ Iteration using element directly: `for n in lst:`

Building lists

Python supports several ways of building a list besides enumerating all elements

```
odd = [1,3,5]
even = [2,4,6]
zeros = 3*[0]                                # List multiplication

lst = odd + even + zeros                      # List concatenation
print(lst)                                   # Output: [1, 3, 5, 2, 4, 6, 0, 0, 0]

lst += [10]
print(lst)                                   # Output: [1, 3, 5, 2, 4, 6, 0, 0, 0, 10]

for i in range(100,141,10):
    odd += [i]
print(odd)                                   # [1, 3, 5, 100, 110, 120, 130, 140]
```

- ▶ Hence, we can construct new lists by adding two (or more) lists
- ▶ Very much like string concatenation and string multiplication. You will see that strings and lists have a lot of properties in common.

Example with list methods

The list class comes with several methods

```
animals = ['dog', 'cat', 'rabbit', 'wolf']

animals.append('tiger') # Add 'tiger' at the end of the list
print(animals)          # ['dog', 'cat', 'rabbit', 'wolf', 'tiger']

animals.insert(0, 'fox') # Insert 'fox' at position 0
print(animals)          # ['fox', 'dog', 'cat', 'rabbit', 'wolf', 'tiger']

animals.remove('rabbit') # Remove first instance of 'rabbit'
print(animals)          # ['fox', 'dog', 'cat', 'wolf', 'tiger']

animals.pop(1).         # Remove element at position 1
print(animals)          # ['fox', 'cat', 'wolf', 'tiger']

animals.sort()          # Sort alphabetically
print(animals)          # ['cat', 'fox', 'tiger', 'wolf']
```

All these methods manipulates (changes) the list content.

More list methods

List methods in addition to `append`, `insert`, `remove`, `pop`, and `sort`

- ▶ `count()`: Returns the number of elements in the list
- ▶ `index()`: Returns the position where `n` first occurs
- ▶ `reverse()`: Reverses the order of the elements in the list
- ▶ `copy()`: Returns a copy of the list (a new list)
- ▶ `clear()`: Removes all elements from the list
- ▶ `extend( list 2)`: Appends `list2` to this list

Example starting with an empty list

```
from random import randint

numbers = []                                # We start with an empty list
for i in range(10):
    rn = randint(1,100)
    numbers.append( rn )                    # Append one element at the time
print( numbers )                           # [26, 90, 77, 82, 30, 48, 100, 85, 55, 88]

numbers.reverse()                           # Reverse order of element
print( numbers )                           # [88, 55, 85, 100, 48, 30, 82, 77, 90, 26]

numbers.sort()                              # Sort in ascending order
print( numbers )                           # [26, 30, 48, 55, 77, 82, 85, 88, 90, 100]

numbers.sort(reverse = True)                # Sort in descending order
print( numbers )                           # [100, 90, 88, 85, 82, 77, 55, 48, 30, 26]
```

- ▶ We start with an empty list (`numbers = []`) and add new random numbers one at the time (`numbers.append( rn )`)
- ▶ By overriding default `reverse = True` in `sort` we change the sorting order

A 10 minute break?

Sequences

Strings and lists are both **sequences** and have a lot in common

```
s = "abcdef"
```

```
print( len(s) )      # 5
print( max(s) )      # e
print( min(s) )      # a
print( s[3] )         # d
print( s[1:3] )       # bc
```

```
for c in s:
    print(c, end=" ") # a b c d e
print()
```

```
a = [1,2,3,4,5,6]
```

```
print( len(a) )      # 5
print( max(a) )      # 5
print( min(a) )      # 1
print( a[3] )         # 4
print( a[1:3] )       # [2, 3]
```

```
for n in a:
    print(n, end=" ") # 1 2 3 4 5
print()
```

- ▶ Something that works for strings often works for list.
- ▶ However, certain things doesn't make sense in both cases, for example
 - ▶ `split()` doesn't make sense for a list
 - ▶ `sum()` doesn't make sense for a string
- ▶ String objects are **immutable** $\Rightarrow$ can't be modified once created
- ▶ List objects are **mutable** $\Rightarrow$ can be modified after creation

Slicing sequences

Accessing certain parts using slicing works for all sequences

- ▶ Accessing certain parts using slicing works for all sequences
- ▶ Similar to range a slice looks like `[start: stop: step]`
- ▶ ... where all of them has certain default values
- ▶ Default values: `start = 0`, `stop = len(...)+1`, `step = 1`
- ▶ Remember that `stop` is not included when used
- ▶ Example: Various slices for list `a = [0,1,2,3,4,5,6,7,8,9]`

```
a[2:5] ==> [2, 3, 4]
```

```
a[2:9:2] ==> [2, 4, 6, 8]
```

```
a[6:2:-1] ==> [6, 5, 4, 3]
```

```
a[:6:] ==> [0, 1, 2, 3, 4, 5] (Uses default for start and step)
```

```
a[5::] ==> [5, 6, 7, 8, 9] (Uses default for stop and step)
```

```
a[:] ==> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] (All default ==> list copy)
```

```
a[::-1] ==> [9, 8, 7, 6, 5, 4, 3, 2, 1, 0] (Reverse copy)
```

- ▶ Remember that it also works for strings
- ▶ `a[::-1]` looks rather cryptic but is frequently used to reverse sequences

Example: Reversing strings

Two variants to reverse a string

```
def reverse(s):  
    rev = ""  
    for c in s:    # Add characters in reverse order  
        rev = c + rev  
    return rev  
  
# Program starts  
s = "Python"  
rev1 = reverse(s)    # Call function reverse(s)  
print(rev1)          # Output: nohtyP  
  
rev2 = s[::-1]       # Slicing  
print(rev2)          # Output: nohtyP
```

- ▶ Version 1: We build a new string by adding the characters in reverse order
- ▶ Version 2: We apply the slice `s[::-1]` $\Rightarrow$ the entire string (start = 0, stop = `len(s)+1`) in reverse order (step = -1)

Search using keyword in

```
def contains(s,x):      # True iff string s contains character x
    for c in s:
        if c == x:
            return True
    return False

# Program starts
s = "Python"
c = 'y'
if contains(s,c):
    print(s, "contains", c) # Output: Python contains y

if c in s:      # Search for char c in string s
    print(s, "contains", c) # Output: Python contains y

a = [1,2,3,4,5]
n = 3
if n in a:      # Search for number n in list a
    print(a, "contains", n) # Output: [1, 2, 3, 4, 5] contains 3
```

Hence, the keyword `in` can also be used to search for elements in a sequence

Convert ranges and strings to lists

The function `list()` can convert strings and ranges to lists

```
a = list("Hello")
print( a, type(a) )      # Output: ['H', 'e', 'l', 'l', 'o'] <class 'list'>

b = list( range(1,6) )
print( b, type(b) )      # Output: [1, 2, 3, 4, 5] <class 'list'>
```

- ▶ `list()` is a conversion function just like `int()`, `float()`, `str()`, and `bool()`
- ▶ `list(x)` tries to convert `x` into a list
- ▶ `list(...)` works for strings and ranges and a few other constructs

List Element Removal

Previously, using list class methods

```
animals = ['dog', 'cat', 'rabbit', 'wolf']

animals.remove('rabbit') # Remove first instance of 'rabbit'
print(animals)           # ['dog', 'cat', 'wolf']

animals.pop(1).           # Remove element at position 1
print(animals)           # ['dog', 'wolf']
```

Using keyword `del`:

```
lst = list( range(10) )
print(lst)      # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

del lst[1]       # Delete element at position 1
print(lst)      # [0, 2, 3, 4, 5, 6, 7, 8, 9]

del lst[3:6]     # Delete positions 3 to 5
print(lst)      # [0, 2, 3, 7, 8, 9]
```

- ▶ The keyword `del` can be used to delete elements or slices from a list
- ▶ It can also be used to remove elements from other types of data structures

Splitting strings using split()

```
s = input("Enter a few whitespace separated words: ")
words = s.split()
print(words)

s = input("Enter a few comma-separated words: ").split(",")
print(words)
```

Usage

```
Enter a few whitespace separated words: Do Re Mi Fa So La
['Do', 'Re', 'Mi', 'Fa', 'So', 'La']
Enter a few comma-separated words: Do,Re,Mi,Fa,So,La
['Do', 'Re', 'Mi', 'Fa', 'So', 'La']
```

- ▶ We can split a string into a list of words using the string method `split()`
- ▶ `split()` uses by default whitespace (" ") to separate words, ...
- ▶ ... but can be configured to use other strings (e.g. `split(",")`)

Applying functions to lists

Three variants for applying function $f(x) = x^2$ to all elements of a list

```
def square_list(a):  
    sq = []  
    for n in a:  
        sq.append( n*n )  
    return sq  
  
def square(x): return x*x  
  
# Program starts  
lst = [1,2,3,4,5]  
sq = square_list(lst)  
print(sq)           # Output: [1, 4, 9, 16, 25]  
  
# Using list comprehensions  
sq = [square(p) for p in lst]  
print(sq)           # Output: [1, 4, 9, 16, 25]  
  
sq = [p*p for p in lst]  
print(sq)           # Output: [1, 4, 9, 16, 25]
```

List comprehensions

```
from math import sqrt

lst = list( range(1,6) )
print(lst)          # [1, 2, 3, 4, 5]

square = [n*n for n in lst]
print(square)       # [1, 4, 9, 16, 25]

root = [round(sqrt(n),2) for n in lst]
print(root)         # [1.0, 1.41, 1.73, 2.0, 2.24]
```

- ▶ `[n*n for n in lst]` is a **list comprehension**
- ▶ We apply the function `n*n` on all elements in list `lst`
- ▶ The result is a new list
- ▶ They are a compact version of iterating over all elements and applying the function on each element.

Conditional list comprehensions

```
# Integers dividable by 7 in range 1 to 50
div_7 = [n for n in range(1,51) if n%7==0]
print(div_7)

# Square all integers, remove everything else
lst = ["ABC", 23.4, 7, True, 9, "xyz", 10]
only_ints = [pow(x,2) for x in lst if type(x) == int]
print(only_ints)
```

Output

```
[7, 14, 21, 28, 35, 42, 49]
[49, 81, 100]
```

- ▶ We can add an if clause to list comprehensions to filter the content
- ▶ Only elements fulfilling the if criteria are added to list
- ▶ `type(x) == int` $\Rightarrow$ `type` is an entity that can be used in boolean expressions

Read multiple integers

```
# Read multiple space separated integers from keyboard
text = input("Enter integers separated by one whitespace: ")
words = text.split()
ints = [int(w) for w in words]

print(f"Largest number is {max(ints)}, smallest is {min(ints)}")
```

Usage

```
Enter integers separated by one whitespace: 23 100 65 97 8 12
Largest number is 100, smallest is 8
```

1. We read input as a single string "23 100 65 97 8 12"
2. We split the string into a list of words
["23", "100", "65", "97", "8", "12"]
3. We convert each word (e.g. "23") to an integer (e.g. 23)
4. We find smallest/largest element by applying min/max on the integer list

Two-dimensional lists (Matrix)

```
# A two-dimensional list
a = [ [1,2,3], [4,5,6], [7,8,9] ] # Format is 3 x 3

print(a)          # [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(a[0][2])    # 1st row, 3rd column ==> 3
print(a[1])       # Entire 2nd row ==> [4,5,6]

a[2][2] = 99      # Replace 9 with 99
print(a)          # [[1, 2, 3], [4, 5, 6], [7, 8, 99]]

# A 4x3 matrix with only 1 elements
b = [4*[1], 4*[1], 4*[1]]
print(b)          # [[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]]
```

- ▶ A two-dimensional list is called a matrix
- ▶ It is a list containing other lists
- ▶ We access individual elements using `a[0][2]`

Simple list programming

Exercise: Write a program `random_elements.py` that:

- ▶ Creates a list containing 10 random floats in interval $[-10,10]$
- ▶ Converts the list to an integer list (correctly rounded off)
- ▶ Prints the smallest and largest elements in the integer list

random_elements.py (Version 1)

```
import random

# A list with ten random floats
floats = []
for i in range(10):
    rnd = random.uniform(-10,10)
    floats.append(rnd)

# Correctly rounded off integers
ints = []
for f in floats:
    ints.append( round(f) )

# Print largest and smallest
lrg = max(ints)
sml = min(ints)
print(f"\nLargest element is {lrg}, smallest is {sml}")
```

We use `append()` repeatedly to build our lists.

random_elements.py (Version 2)

A much shorter version using list comprehensions

```
import random as rd

# Ten random floats
floats = [rd.uniform(-10,10) for i in range(10)]

# Rounded of integers
ints = [round(f) for f in floats]

# Print largest and smallest
print(f"\nLargest element is {max(ints)}, smallest is {min(ints)}")
```

Which version is the best? Version 1 or 2?

Lists - Summary

- ▶ A list is a sequential data structure
- ▶ Sequential $\Rightarrow$ all elements have a position, we have a first and last element
- ▶ Lists are mutable $\Rightarrow$ we can manipulate (add, remove, swap) the list elements
- ▶ Lists are very flexible $\Rightarrow$ many different ways to create and manipulate them
- ▶ List and strings are both sequences $\Rightarrow$ many properties in common
- ▶ **Lists are great $\Rightarrow$ we use them a lot $\Rightarrow$ get familiar with them!**