Session 6

1DV501 Intro to programming

- Software components are used like hardware components.
- A software system can be built largely by assembling pre-existing software building blocks.
- The simplest of these is the function (textbook Chapter 6 and Chapter 7)
- A more powerful technique uses software objects.

Python is object oriented

- An Object Oriented programming language allows the programmer to define, create, and manipulate objects.
- Objects bundle together data and functions. Like other variables, each Python object has a type, or class.
- The terms `class` and `type` are synonymous.

- An `object` is an instance of a `class` `
- An object's data consists of its `instance variables`
- object is an instance of a class

In [2]:
```
a = 1
```

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)

class bank_account	Object 1	Object 2	Object 3
Owner:	Jonas	Henrik	Nils
No:	4758-8696	3246-9744	5432-2347
Balance:	34.345 kr	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
```

In [4]:
```
s = "Hello"        # "Hello" is an object of type/class string

print(s.upper())     # Apply method on string object "Hello"
```

```
HELLO
```

In [7]:
```
s.isdecimal()
```

Out[7]:
```
False
```

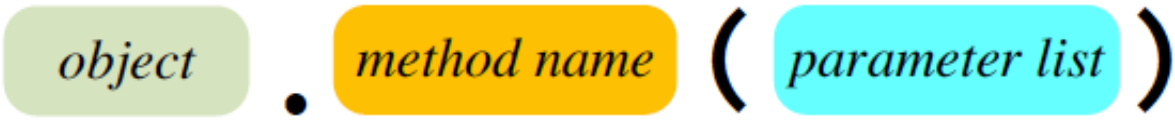
- A class defines properties of a given type of objects
- A class definition (often a separate file) is a bit of code defining:
  - Attributes: The data we associate with the class (for example owner, account number, and saldo for a back account)
  - Methods: Operations we can do on an object (for example `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 = "My name is Sir Lancelot of Camelot."
print( s.upper() )
```

- General pattern for a method call



- object is an expression that represents object, above is a reference to a string object.
- The **period**, pronounced **dot**, associates an object expression with the method to be called.
- **methodname** is the name of the method to execute.
- The **parameterlist** is comma-separated list of parameters to the **method**. For some methods the parameter list may be empty, but the **parentheses always are required**.
- In this lecture we look at a few common classes from the Python library
- We will *not* create our own classes

String objects and methods

The string class `str`

- Strings are objects of a predefined class `str

```
print( type("What… is your favourite colour?") )      # Output: <cla
ss 'str'>
```

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

```
s = "Blue. Right. Off you go."

print( s.upper() )
print( s.count("l") )
print( s.find("lo") )
print( s.endswith("xxx") )
print( s.isalpha() )
```

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

In [12]:
```
s = "Blue. Right. Off you go."

print( s.upper() )
print( s.count("f") )
print( s.find("e.") )
print( s.endswith("xxx") )
print( s.isalpha() )

s = "Arthur"

print( s.isalpha() )
```

```
BLUE. RIGHT. OFF YOU GO.
2
3
False
False
True
```

The Python Standard Library

- The string class `str` comes with many methods
- It is hard to remember all details about all method`
- The official documentation for the string class is:
- https://docs.python.org/3/library/ (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
- **This 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)` -> first character in alphabetical order (Returns a character which is alphabetically the lowest character in the string.)
- `max(s)` -> last character in alphabetical order
- alphabetical order: First digits, than upper case, then lower case, other character are sorted based on their ASCII number
- Notice also how built-in functions are applied (e.g. `len(s)`) compared to how methods are applied (e.g. `s.upper()`)

In [13]:
```
s = "abcABC"

print( len(s) )
print( min(s) )
print( max(s) )
print( min("aA1"))
print( min("aA 1{"))
print( max("aA 1{"))
```

```
6
A
c
1

{
```

In [14]:
```
s = 'What is the air-speed velocity of an unladen swallow?'

print( len(s) )
print( min(s) )
print( max(s) )
```

```
53

y
```

In [16]:
```
ord('a')
```

Out[16]:
```
97
```

fractions — Rational numbers

Source code: Lib/fractions.py

https://docs.python.org/3/library/fractions.html (https://docs.python.org/3/library/fractions.html)

The fractions module provides support for rational number arithmetic.

A **Fraction instance** can be **constructed** from a pair of integers, from another rational number, or from a string.

```
class fractions.Fraction(numerator=0, denominator=1)
class fractions.Fraction(other_fraction)
class fractions.Fraction(float)
class fractions.Fraction(decimal)
class fractions.Fraction(string)
```

The first version requires that numerator and denominator are *instances* of numbers. Rational and returns a new Fraction instance with value numerator/denominator.

In [18]:
```
from fractions import Fraction

Fraction(0.5)
```

Out[18]:
```
Fraction(1, 2)
```

Fraction objects and methods

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` -> the data values representing the object state

https://docs.python.org/3/library/fractions.html (https://docs.python.org/3/library/fractions.html)

In [19]:
```
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
```

```
1/2 <class 'fractions.Fraction'>
5/6
5
6
```

In [26]:
```
fsum.
```

Out[26]:
```
5
```

`limit_denominator(max_denominator=1000000)`

Finds and returns the closest Fraction to self that has denominator at most *max_denominator*.

- This method is useful for finding rational approximations to a given floating-point number:

In [27]:
```
from fractions import Fraction
Fraction('3.1415926535897932').limit_denominator(10)
```

Out[27]:
```
Fraction(22, 7)
```

In [32]:
```
from math import pi
Fraction(pi).limit_denominator(100000)
```

Out[32]:
```
Fraction(312689, 99532)
```

In [30]:
```
pi-(22/7)
```

Out[30]:
```
-0.0012644892673496777
```

In [33]:
```
f1 = Fraction(6,345)
f2 = Fraction(53/212)
print((f1+f2))
print((f1*f2))
```

```
123/460
1/230
```

In [ ]:
```
Fraction.from_float(0.3) # From Python 3.2 onwards, you can also co
nstruct a Fraction instance directly from a float.
```

- 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

Working with lists

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

  1. Data should be ordered
  2. Not the same element twice
  3. Important that look-up is fast
  4. In general: Important that operations X,Y,Z are fast

- Selecting data structure is a design decision -> 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` -> `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 -> no need for any import statemen`

In [37]:
```
min_lista = [1,2,546564,3,4.5,'en string']
```

In [41]:
```
print(min_lista[:-1])
```

```
[1, 2, 546564, 3, 4.5]
```

In [43]:
```
min_andra_lista = [0,2,3,min_lista,f1]
```

In [46]:
```
print(min_andra_lista[3][1])
```

```
2
```

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:`

In [47]:
```
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()
```

```
[1, 2, 99, 4, 5]
1 2 99 4 5
1 2 99 4 5
```

In [48]:
```
my_list = [1,3,4,'Sir Lancelot of Camelot',2.5]

if 3 in my_list: print('Yes, there is a three')
if not 'King Arthur' in my_list: print('And not King Arthur')
if 'Lancelot of Camelot' in my_list: print('And Sir Lancelot of Cam
elot') # note this
```

```
Yes, there is a three
And not King Arthur
```

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)

lst += [10]
print(lst)

for i in range(100,141,10):
    odd += [i]
print(odd)
```

- 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.

*the action of linking things together in a series

In [49]:
```
odd = [1,3,5]
even = [2,4,6]
zeros = 3*[0]                  # List multiplication

lst = odd + even + zeros     # List concatenation*
print(lst)
```

```
[1, 3, 5, 2, 4, 6, 0, 0, 0]
```

In [51]:
```
lst += [10]
print(lst)
```

```
[1, 3, 5, 2, 4, 6, 0, 0, 0, 10, 10]
```

In [52]:
```
for i in range(100,141,10):
    odd += [i]
print(odd)
```

```
[1, 3, 5, 100, 110, 120, 130, 140]
```

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)

animals.insert(0,'fox')  # Insert 'fox' at position 0
print(animals)

animals.remove('rabbit') # Remove first instance of 'rabbit'
print(animals)

animals.pop(1)           # Remove element at position 1
print(animals)

animals.sort()           # Sort alphabetically
print(animals)
```

- All these methods manipulates (changes) the list content.

In [53]:
```
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']
```

```
['dog', 'cat', 'rabbit', 'wolf', 'tiger']
['fox', 'dog', 'cat', 'rabbit', 'wolf', 'tiger']
['fox', 'dog', 'cat', 'wolf', 'tiger']
['fox', 'cat', 'wolf', 'tiger']
['cat', 'fox', 'tiger', 'wolf']
```

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

In [54]:
```
animals.extend(animals)
print(animals)
```

```
['cat', 'fox', 'tiger', 'wolf', 'cat', 'fox', 'tiger', 'wolf']
```

In [55]:
```
animals.count('cat')
```

Out[55]: 2

In [57]:
```
animals2 = animals
```

In [60]:
```
animals.pop(0)
```

Out[60]:
```
'cat'
```

In [63]:
```
animals2.pop(0)
```

Out[63]:
```
'fox'
```

In [64]:
```
animals
```

Out[64]:
```
['tiger', 'wolf', 'cat', 'fox', 'tiger', 'wolf']
```

In [65]:
```
## Why not just assign the list to a new name? Not copy???

animals_copy = animals.copy()

## or ?

animals2 = animals
```

In [ ]:
```

```

In [66]:
```
print(len(animals_copy))
print(len(animals2))
```

```
6
6
```

In [67]:
```
animals.pop(4)
```

Out[67]:
```
'tiger'
```

In [68]:
```
print(len(animals_copy))
print(len(animals2))

#By assignment you are getting different names for the same object.
```

```
6
5
```

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 )

numbers.reverse()           # Reverse order of element
print( numbers )

numbers.sort()              # Sort in ascending order
print( numbers )

numbers.sort(reverse = True) # Sort in descending order
print( numbers )
```

- 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

In [69]:
```
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('Random: \t', numbers )           #


numbers.reverse()           # Reverse order of element
print('Reverse: \t', numbers )          #

numbers.sort()              # Sort in ascending order
print('Ascending: \t', numbers )        #

numbers.sort(reverse = True) # Sort in descending order
print('Descending: \t', numbers )       #
```

```
Random:         [1, 78, 86, 99, 74, 47, 45, 62, 44, 84]
Reverse:        [84, 44, 62, 45, 47, 74, 99, 86, 78, 1]
Ascending:      [1, 44, 45, 47, 62, 74, 78, 84, 86, 99]
Descending:     [99, 86, 84, 78, 74, 62, 47, 45, 44, 1]
```

Slicing sequences

Sequences

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

String

```
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()
```

List

```
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

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)

rev2 = s[::-1]     # Slicing
print(rev2)
```

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

In [70]:
```
def reverse(s):
    rev = ""
    for c in s:    # Add characters in reverse order
        rev = c + rev
    return rev
```

In [71]:
```
reverse('Brother Maynard — bring forth the holy hand grenade!')
```

Out[71]:
```
'!edanerg dnah yloh eht htrof gnirb — dranyaM rehtorB'
```

In [72]:
```
'Brother Maynard — bring forth the holy hand grenade!'[::-1]
```

Out[72]:
```
'!edanerg dnah yloh eht htrof gnirb — dranyaM rehtorB'
```

Search using keyword `in`

```
def contains(s,l):
    for c in s:
        if c == l:
            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

In [73]:
```
def contains(s,l):
    for c in s:
        if c == l:
            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
```

```
Python contains y
Python contains y
[1, 2, 3, 4, 5] contains 3
```

In [75]:
```
s = 'Hello'
if 'H' in s: print('Yes')
```

```
Yes
```

Convert ranges and strings to lists

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

```
a = list("Nobody expects the Spanish Inquisition")
print( a, type(a) )

b = list( range(1,6) )
print( b, type(b) )
```

- `list()` is a convertion function 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 contsructs

In [76]:
```
a = list("Nobody expects the Spanish Inquisition")
print( a, type(a) )     # Output: ['H', 'e', 'l', 'l', 'o'] <class 'list'>
```

```
['N', 'o', 'b', 'o', 'd', 'y', ' ', 'e', 'x', 'p', 'e', 'c', 't', 's', ' ', 't', 'h', 'e', ' ', 'S', 'p', 'a', 'n', 'i', 's', 'h', ' ', 'I', 'n', 'q', 'u', 'i', 's', 'i', 't', 'i', 'o', 'n'] <class 'list'>
```

In [77]:
```
b = list( range(1,6) )
print( b, type(b) )
```

```
[1, 2, 3, 4, 5] <class 'list'>
```

Splitting strings using `split()`

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

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

- Usage

```
Enter a few words: Spam! Spam! Spam! Spam! Spam! Spam!
['Spam!', 'Spam!', 'Spam!', 'Spam!', 'Spam!', 'Spam!']

Enter a few comma-separated words: Egg,Sausage,Spam,Egg,Spam
['Egg', 'Sausage', 'Spam', 'Egg', 'Spam']
```

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

In [78]:
```
s = input("Enter a few words: ")
words = s.split()
print(words)

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

```
Enter a few words: Här skriver jag olika saker
['Här', 'skriver', 'jag', 'olika', 'saker']
Enter a few comma-separated words: Med,komma,tecken
['Med', 'komma', 'tecken']
```

List comprehension

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]
```

In [79]:
```
def square_list(a):
    sq = []
    for n in a:
        sq.append( n*n )
    return sq
```

In [89]:
```
square(10)
```

Out[89]: 100

In [88]:
```
[square(p) for p in min_nummer_lista]
```

Out[88]: [4, 16, 36, 100]

In [91]:
```
[x**x for x in min_nummer_lista[:-1]]
```

Out[91]: [4, 256, 46656]

In [ ]:
```
```

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

print(square_list(lst))

##

sq = square_list(lst)
print(sq)
```

In [ ]:
```
# Using list comprehensions
sq = [square(p) for p in lst]
print(sq)
```

In [ ]:
```
sq = [p*p for p in lst]
print(sq)
```

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)
```

- `[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.

Advanced 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 \verb+if+ clause to list comprehensions to filter the content
- Only elements fulfilling the if criteria are added to list
- `type(x) == int` -> type is an entity that can be used in boolean expressions

In [92]:
```
[n for n in range(1,51) if n%7==0]
```

Out[92]:
```
[7, 14, 21, 28, 35, 42, 49]
```

In [ ]:
```
# 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)
```

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

In [93]:
```
# 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)}")
```

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

Two-dimensional lists (Matrix)

```
# A two-dimensional list
a = [ [1,2,3], [4,5,6], [7,8,9] ] # Format is 3x3

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]`

In [ ]:
```
a = [ [1,2,3], [4,5,6], [7,8,9] ] # Format is 3x3
```

In [ ]:
```
a[0]
```

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

In [ ]:
```
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) )  # NOTE We use append() repeatedly to b
uild our lists.


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

A much shorter version using list comprehensions

(Version 2)

In [ ]:
```
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)}")
```

In [ ]:
```

```