Fundamentals of programming

Lecture 2. Procedural programming

- Structured types
- What is a function
- How to write functions

Structured types

- List
- Tuple
- Dictionary

Lists

Lists represent the finite ordered sets indexed by non-negative numbers. See [3, sections 3.1.4, 5.1].
Operations:

- creation
- accessing values (index, len), changing values (**lists are mutable**)
- removing items (pop), inserting items (insert)
- slicing
- nesting
- generate list using range(), list in a for loop
- lists as stacks (append, pop)

```
# create
a = [1, 2, 'a']
print (a)
x, y, z = a
print(x, y, z)

# indices: 0, 1, ..., len(a) - 1
print a[0]
print ('last element = ', a[len(a)-1])

# lists are mutable
a[1] = 3
print a
```

```
# slicing
print a[:2]
b = a[:]
print (b)
b[1] = 5
print (b)
a[3:] = [7, 9]
print(a)
a[:0] = [-1]
print(a)
a[0:2] = [-10, 10]
print(a)
```

```
# lists as stacks
stack = [1, 2, 3]
stack.append(4)
print stack
print stack.pop()
print stack
```

```
# nesting
c = [1, b, 9]
print (c)
```

```
#generate lists using range
l1 = range(10)
print l1
l2 = range(0,10)
print l2
l3 = range(0,10,2)
print l3
l4 = range(9,0,-1)
print l4
```

```
#list in a for loop
l = range(0,10)
for i in l:
    print i
```

Tuples

Tuples are immutable sequences. A **tuple** consists of a number of values separated by commas. See [3, section 5.3].
Operations:

- packing values (creation)
- nesting
- empty tuple
- tuple with one item
- sequence unpacking

```
# Tuples are immutable sequences
# A tuple consists of a number of
values separated by commas

# tuple packing
t = 12, 21, 'ab'
print(t[0])

# empty tuple (0 items)
empty = ()
```

```
# tuple with one item
singleton = (12,)
print (singleton)
print (len(singleton))

#tuple in a for
t = 1,2,3
for el in t:
    print el
```

```
# sequence unpacking
x, y, z = t
print (x, y, z)
```

```
# Tuples may be nested
u = t, (23, 32)
print(u)
```

Dictionaries

A **dictionary** is an unordered set of (key, value) pairs with unique keys. The keys must be immutable. See [3, section 5.5]
Operations:

- creation
- getting the value associated to a given key
- adding/updating a (key, value) pair
- removing an existing (key, value) pair
- checking whether a key exists

```
#create a dictionary
a = {'num': 1, 'denom': 2}
print(a)

#get a value for a key
print(a['num'])
```

```
#set a value for a key
a['num'] = 3
print(a)
print(a['num'])
```

```
#delete a key value pair
del a['num']
print (a)
```

```
#check for a key
if 'denom' in a:
    print('denom = ', a['denom'])
if 'num' in a:
    print('num = ', a['num'])
```

Identity, value and type

Recall what is a *name* and an *object* (*identity*, *type*, *value*).

- mutable objects: lists, dictionaries, sets
- immutable: numbers, strings, tuples

Determine the identity and the type of an object using the built-in functions:

- id(object)
- type(object), isinstance(object, type)

Procedural programming

A **programming paradigm** is a fundamental style of computer programming.

Imperative programming is a programming paradigm that describes computation in terms of statements that change a program state.

Procedural programming is imperative programming in which the program is built from one or more procedures (also known as subroutines or functions).

What is a function

A **function** is a self contained block of statements that:

- has a *name*,
- may have a list of (formal) *parameters*,
- may *return* a value
- has a *documentation* (specification) which consists of:
    - a *short description*
    - *type and description for the parameters*
    - conditions imposed over the input parameters (*precondition*)
    - type and description for the return value
    - conditions that must be true just after the execution (*post-condition*).
    - Exceptions

```
def max(a, b):
    """
    Compute the maximum of 2 numbers
    a, b - numbers
    Return a number - the maximum of two integers.
    Raise TypeError if parameters are not integers.
    """
    if a>b:
        return a
    return b
def isPrime(a):
    """
      Verify if a number is prime
      a an integer value  (a>1)
      return True if the number is prime, False otherwise
    """
```

Functions

The following function is working but we do not consider as a function at the lab/exam

```
def f(k):
    l = 2
    while l<k and k % l>0:
        l=l+1
    return l>=k
```

Every function written by you should:

- use meaningful names (function name, variable names)
- provide specification
- include comments
- have a test function (see later)

```
def isPrime(nr):
    """
      Verify if a number is prime
      nr - integer number, nr>1
      return True if nr is prime, False otherwise
    """
    div = 2  #search for divider starting from 2
    while div<nr and nr % div>0:
        div=div+1
    #if the first divider is the number itself than the number is prime
    return div>=nr;
```

Definition

A **function definition** is an executable statement introduced using the keyword **def**.

The function definition does not execute the function body; this gets executed only when the function is called.
A function definition defines a user-defined function object.

```
def max(a, b):
    """
    Compute the maximum of 2 numbers
    a, b - numbers
    Return a number - the maximum of two integers.
    Raise TypeError if parameters are not integers.
    """
    if a>b:
        return a
    return b
```

Variable scope

A *scope* defines the visibility of a name within a block.
If a local variable is defined in a block, its scope includes that block.
All variables defined at a particular indentation level or scope are considered local to that indentation level or scope

- Local variable
- Global variable

```
global_var = 100

def f():
    local_var = 300
    print local_var
    print global_var
```

Rules to determine the scope of a particular name (variable, function name):

- a name defined inside a block is visible only inside that block
- formal parameters belong to the scope of the function body (visible only inside the function)
- name defined outside the function (at the module level) is belong to the module scope

Variable scope

When a name is used in a code block, it is resolved using the nearest enclosing scope.

```
a = 100
def f():
    a = 300
    print a

f()
print a
```

```
a = 100
def f():
    global a
    a = 300
    print a

f()
print a
```

At any time during execution, names are resolved using :

- the innermost scope, which is searched first, contains the local names (inside the block)
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

globals() locals() - python built in functions for inspecting global/local variables

```
a = 300
def f():
    a = 500
    print a
    print locals()
    print globals()

f()
print a
```

Calls

A ***block*** is a piece of Python program text that is executed as a unit. Blocks of code are denoted by line indentation.
A **function body** is a block. A block is executed in an *execution frame*. When a function is invoked a new execution frame is created.

```
max(2,5)
```

An execution frame contains:

- some administrative information (used for debugging)
- determines where and how execution continues after the code block's execution has completed
- defines two namespaces, the local and the global namespace, that affect execution of the code block.

A *namespace* is a mapping from names (identifiers) to objects.

A particular namespace may be referenced by more than one execution frame, and from other places as well.

Adding a name to a namespace is called binding a name (to an object);
changing the mapping of a name is called rebinding;
removing a name is unbinding.

Namespaces are functionally equivalent to dictionaries (and often implemented as dictionaries).

Passing parameters

Formal parameters is an identifier for an input parameter of a function. Each call to the function must supply a corresponding value (argument) for each mandatory parameter
Actual parameter is a value provided by the caller of the function for a formal parameter.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called (arguments are passed *by object reference*)

```
def change_or_not_immutable(a):
    print ('Locals ', locals())
    print ('Before assignment: a = ', a, ' id = ', id(a))
    a = 0
    print ('After assignment: a = ', a, ' id = ', id(a))

g1 = 1      #global immutable int
print ('Globals ', globals())
print ('Before call: g1 = ', g1, ' id = ', id(g1))
change_or_not_immutable(g1)
print ('After call: g1 = ', g1, ' id = ', id(g1))
```

```
def change_or_not_mutable(a):
    print ('Locals ', locals())
    print ('Before assignment: a = ', a, ' id = ', id(a))
    a[1] = 1
    a = [0]
    print ('After assignment: a = ', a, ' id = ', id(a))

g2 = [0, 1] #global mutable list
print ('Globals ', globals())
print ('Before call: g2 = ', g2, ' id = ', id(g2))
change_or_not_mutable(g2)
print ('After call: g2 = ', g2, ' id = ', id(g2))
```

Test cases

Before implementing a function we write test cases in order to understand what the function must perform. This is also a a design step.

A **test case** specifies a set of test inputs, execution conditions, and expected results that you identify to evaluate a particular part of a program (e.g. a function).

For each function we will create a test function, verify input output pairs using the built in **assert** function

Test Function - Calculator example

1 Feature 1. Add a number to calculator.
2 Running scenario for adding numbers
3 Workitems/Tasks

T1	Compute the greatest common divisor of two integers (see g, i)
T2	Add two rational numbers (see c, e, g, i)
T3	Implement calculator: init, add, and total
T4	Implement user interface

T1 Compute the greatest common divisor of two integers (see g, I)
Test case

Table format

Input: (params a,b)	Output: gdc(a,b)
2 3	1
2 4	2
6 4	2
0 2	2
2 0	2
24 9	3

Test function

```
def test_gcd():
    assert gcd(2, 3) == 1
    assert gcd(2, 4) == 2
    assert gcd(6, 4) == 2
    assert gcd(0, 2) == 2
    assert gcd(2, 0) == 2
    assert gcd(24, 9) == 3
```

Implement gdc

```
def gcd(a, b):
    """
     Compute the greatest common divisor of two positive integers
     a, b integers a,b >=0
     Return the greatest common divisor of two positive integers.
    """
    if a == 0:
        return b
    if b == 0:
        return a
    while a != b:
       if a > b:
           a = a - b
       else:
           b = b - a
    return a
```

How to write functions

Apply test-driven development (TDD) steps

TDD requires developers to create automated unit tests that clarify code requirements before writing the code itself.
When you create a new function (*f*), follow TDD steps:

- Add a test
    - **Define a test function (*test_f*()) which contains test cases written using *assert*ions.**
    - Concentrate on the **specification of *f***.
    - **Define *f*: name, parameters, precondition, post-condition, and an empty body.**
- Run all tests and see if the new one fails
    - Your program may have many functions, so **many test functions**.
    - At this stage, ensure the new **test_f() fails**, while **other test functions pass** (written previously).
- **Write the body of f**
    - Now the specification of f is well written and you concentrate on **implementing the function according to pre/post-conditions** and on **passing all test cases written for *f***.
    - **Do not concentrate on technical aspects (duplicated code, optimizations, etc).**
- Run all tests and see them succeed
    - Now, the developer is confident that the function meets the specification.
    - The final step of the cycle can be performed.
- Refactor code
    - Finally, you must clean up the code using refactorings techniques.

Refactorings

Code refactoring is "disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior" [5].
Code smell is any symptom in the source code of a program that possibly indicates a deeper problem:

- **Duplicated code: identical or very similar code exists in more than one location.**
- **Long method: a method, function, or procedure that has grown too large.**

Refactorings:

- **Extract Method**
  - You have a code fragment that can be grouped together.
  - Turn the fragment into a method whose name explains the purpose of the method.
- **Substitute Algorithm**
  - You want to replace an algorithm with one that is clearer.
  - Replace the body of the method with the new algorithm.
- **Replace Temp with Query**
  - You are using a temporary variable to hold the result of an expression.
  - Extract the expression into a method. Replace all references to the temp with the expression. The new method can then be used in other methods.

References

1. *The Python language reference.* *http://docs.python.org/py3k/reference/index.html*
2. *The Python standard library.* *http://docs.python.org/py3k/library/index.html*
3. *The Python tutorial.* *http://docs.python.org/tutorial/index.html*
4. Kent Beck.*Test Driven Development: By Example. Addison-Wesley Longman, 2002*. See also Test-driven development. http://en.wikipedia.org/wiki/Test-driven_development
5. Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999. See also http://refactoring.com/catalog/index.html