BABEŞ-BOLYAI UNIVERSITY
Faculty of Mathematics and Computer Science

Algorithms and Programming

Lecture 2 – Procedural Programming

Camelia Chira

Last time

- Programming process
  - What is programming?
  - Basic elements of Python
    - Python programs
    - Data types: string, number
    - Variables and expressions
    - Statements: assignments, conditionals, loops

Today

- Continue with data types: string, list, tuple, dictionary

- Software development process
  - Simple feature-driven development

- Programming paradigms

- Functions
  - Definition
  - Call
  - How to write a function

Strings

- *Domain*: character sequence(letters, special characters, digits)
- *Operations*: concatenation, search
- Immutable
- Enclose in quotation marks or single quotes

```
s = "hello there"
```

```
>>> s = "hello there"
>>> s
'hello there'
```

- Concatenate strings

```
name= "Zara"
greeting1 = s + name
greeting2 = s + " " + name
greeting3 = s + " " + name * 3
```

Strings

- Indexing

```
  s    = “abc”
index    0 1 2
index   -3 -2 -1
```

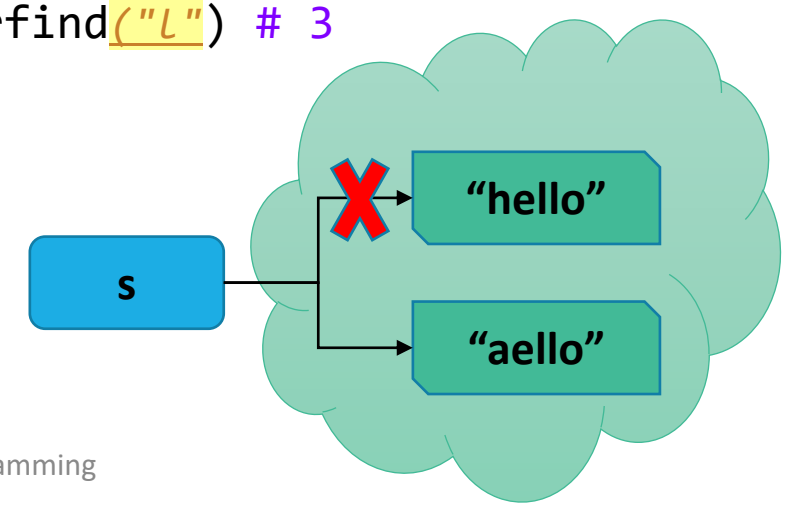
```
s[0], s[-3] -> “a”
s[1], s[-2] -> “b”
s[2], s[-1] -> “c”
```

- Strings are immutable

```
s    = “hello”
s[0]= “a” # gives an error
s    = “a” + s[1:] # ok (s is bound
                   to new object)
```

- Slicing and search

```
s = "hello there"
s[0]     # ’h’
s[2:5]   # ’llo’
s[2:]    # ’llo there’
s[:2]    # ’he’
s[-1]    # ’e’
s.find("l")  # 2
s.rfind("l") # 3
```



Input / Output

- print

```
x = 2
print(x)
x_str = str(x)
print("My number is", x, ".")
print("My number is " + x_str + ". ")
```

- input

```
age = input("Your age is:")
print(age)
```

input gives you a string

```
s = input("Your age is:")
age = int(s)
print("Age is ", age, ". Next year you will be ", age + 1, "...")
```

Lists

- *Domain*: sequence of elements (usually of same type e.g. list of ints, but can be of different types) separated by , and enclosed by [ ]
- Operations:
  - Create (manually, using range)
  - Access (index, len) and modify elements
  - Remove (pop) and insert (insert) elements
  - Slicing and encapsulating
  - Using as stacks (append, pop)
- Mutable (elements of a list can be modified)

Lists: indices and ordering

Create

```
a_list = [] # empty list
b_list = [2, 5, 7]
len(b_list) # evaluates to 3
x, y, z = b_list

c_list = [2, 'a', 3, [1, 5, 4]]
len(c_list) # evaluates to 4
```

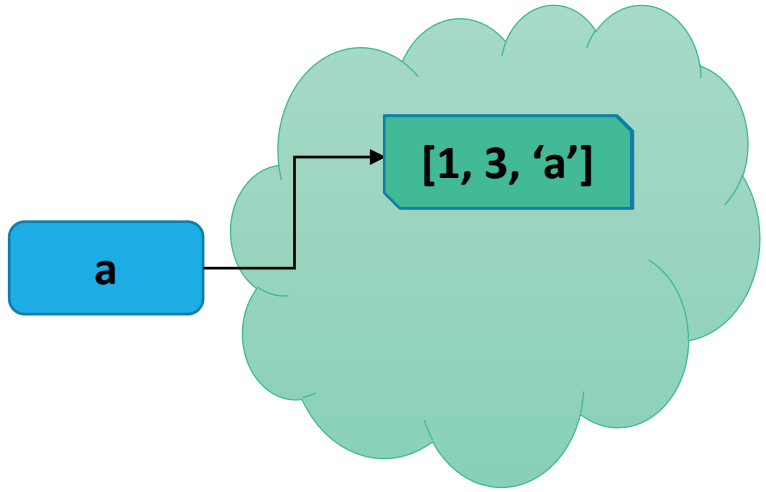
Indices 0, 1,…,len(list)-1

```
c_list[0]     # is 2
c_list[2] + 1 # is 4
c_list[3]     # list [1, 5, 4]
c_list[4]     # error
print("Last element ", c_list[len(c_list)-1])
```

Lists are mutable

```
a = [1, 2, 'a']
a[1] = 3
print(a)
```

a is the same object !



Lists: slicing and range

```
a= [1, 2, 3]
print (a[:2]) # [1, 2]

b = a[:]
print(b)

b[1] = 5
print(b)

a[len(a):] = [7, 9]
print(a) # [1, 2, 3, 7, 9]

a[:0] = [-1]
print(a) # [-1, 1, 2, 3, 7, 9]

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

```
# nesting
c = [1, b, 9]
print(c) # [1, [1, 5, 3], 9]

#generate lists using range
l1 = range(10)
print(l1) # [0, 1, ..., 9]

l2 = range(0,10)
print(l2) # [0, 1, ..., 9]

l3 = range(0,10,2)
print(l3) # [0, 2, 4, 6, 8]

l4 = range(9,0,-1)
print(l4) # [9, 8, 7, ..., 1]
```

Iterating over a list

- Compute the sum of elements in a list

```
total = 0
for i in range(len(my_list)):
    total += my_list[i]
print(total)
```

```
total = 0
for elem in my_list:
    total += elem
print("Sum is ", total)
```

- `my_list = [1, 4, 5, 10, 10]`
- List elements are indexed from `0` to `len(my_list)-1`
- `range(n)` goes from `0` to `n-1`

List operations

- Add elements to the end of the list (mutates the list)

```
a = [1, 2, 3]
a.append(7) # a = [1, 2, 3, 7]
```

- Concatenation of lists

```
a = [1, 2, 3]
b = [4, 5]
c = a + b # c = [1, 2, 3, 4, 5]
          # a and b unchanged

a.extend([10, 11]) # a = [1, 2, 3, 10, 11]
                   # a mutated to new value
```

- Remove

```
a = [0, 1, 2, 1, 3, 4, 5]

# remove element by value, first occurence
a.remove(3) # mutates a
a.remove(1) # mutates a

# remove element by index
del(a[1]) # mutates a, a = [0, 1, 4, 5]

#remove last element in the list
a.pop() # mutates a, returns 5
```

List operations

```
# from strings to lists
s = "Year< 2“
list(s)      # ['Y', 'e', 'a', 'r', '<', ' ', '2']
s.split()    # ['Year<', '2']
s.split("<") # ['Year', ' 2']

# from lists to strings
a = ["a", "b", "c"]
"".join(a)  # 'abc'
" ".join(a) # 'a b c'
```

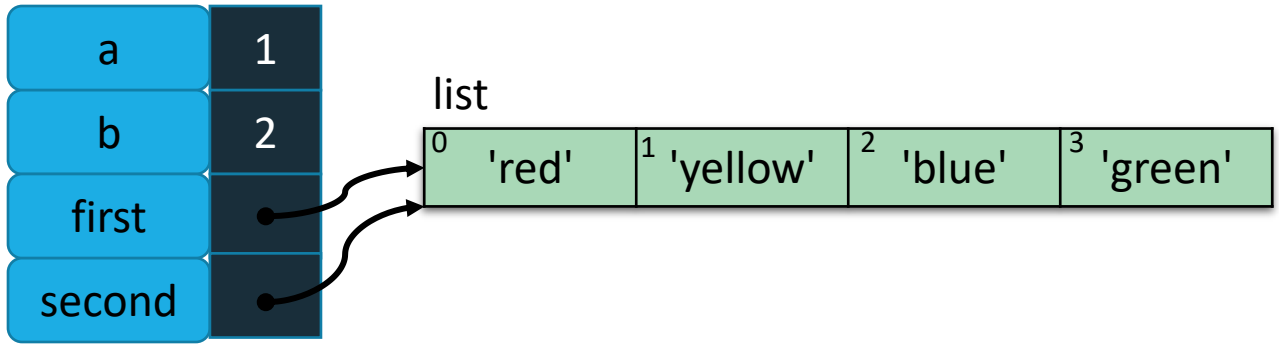
```
# sort and reverse on lists
a = [7, 2, 5, 3]
sorted(a) # returns sorted list, does not mutate a
a.sort() # mutates a = [2, 3, 5, 7], returns nothing
a.reverse() # mutates a = [7, 5, 3, 2]
```

Lists in memory

- Lists are mutable – pay attention to side effects

```
a = 1
b = a
print("a = ", a)
print("b = ", b)
b = b + 1
print(a)
print(b)

first = ["red", "yellow", "blue"]
second = first
second.append("green")
print(first)
print(second)
```

```
a =  1
b =  1
1
2
['red', 'yellow', 'blue', 'green']
['red', 'yellow', 'blue', 'green']
>>>
```



Creating a copy of a list

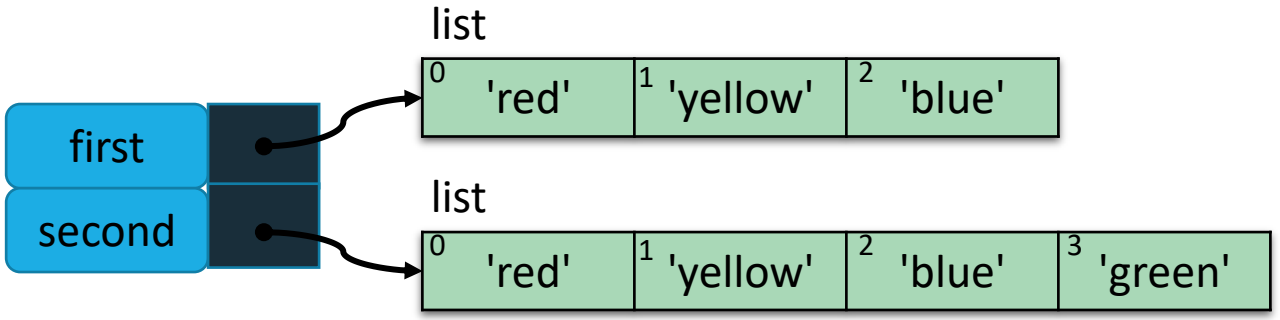
- Cloning a list: create a new list and copy every element

```
second = first[:]
```

Note that: `second = first` is not the same

```
first = ["red", "yellow", "blue"]
second = first[:]
second.append("green")
print(first)
print(second)
```

```
['red', 'yellow', 'blue']
['red', 'yellow', 'blue', 'green']
>>>
```



Tuples

- *Domain*: sequence of values (same or different type) separated by ,
- Operations: creation (tuple packing), unpacking, empty /single element tuples
- Immutable – element values can not be changed

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

# empty tuple (0 items)
empty = ()

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

```
# tuple with one item
singleton = (12,)
len(singleton) # 1

t = 1,2,3
len(t)    # 3
t[0]      # 1
t[1:2]    # (2,)
t + (4, 5) # (1, 2, 3, 4, 5)
t[0] = 0  # Error!

u = t, (4, 5)
print(u) # ((1, 2, 3), (4, 5))
```

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

#can use in swaps
a = 1
b = 2
(a, b) = (b, a)
```

Dictionaries

- *Domain*: sequence of unordered pairs of data (key, value) – with unique keys
- Operations:
  - Creation
  - Access the value for given key
  - Add/modify/delete a given pair (key, value)
  - Verify if a key exists
- Immutable

```
#empty dictionary
d = {}

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

#get a value for a key
a['num'] # 1
```

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

#add an entry
a['type'] = 'f'

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

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

Values:

- Can be duplicates
- Any type

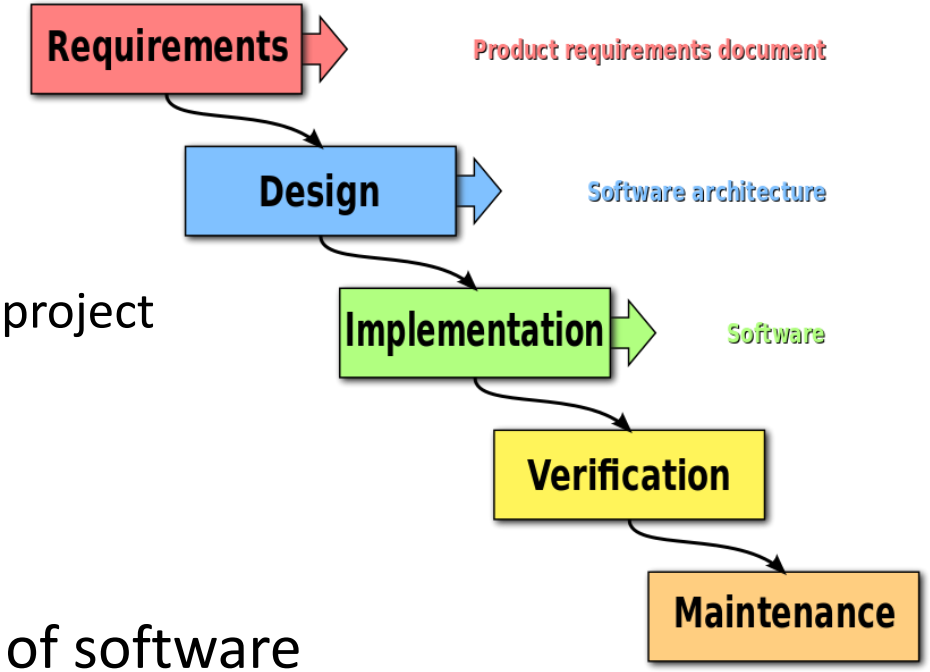
Keys:

- Must be unique
- Immutable type

Software development process

- Roles in software engineering
  - Programmer / software developer
    - Writes and develops programs for users
  - Client
    - The one interested / affected by the results of a project
  - User
    - Runs the program on the computer
- Software development process:
  - Includes creation, launch and maintainence of software
  - Indicate the steps to be followed and their order

The waterfall model



Software development process

- Steps in solving a problem:
  - Problem definition
  - Requirements
  - Use case scenario
  - Identify the features and separate them on iterations
  - Identify the activities or tasks (for each feature) and describe them

Software development process

- Steps in solving a problem:
  - Problem definition
    - A short description of the problem
    - *A teacher (**client**) needs an application for students (**users**) learning how to find the smallest prime number greater than a given natural number n.*
  - Requirements
    - Define what is required from the client perspective (what the application needs to do)
    - Identify input and output data
    - ***Input**: n – a natural number*
    - ***Output**: the smallest prime number greater than n*
  - Use case scenario

Run	1	2	3	4
Input	5	0	11	-3
Output	7	2	13	Please enter a natural number:

Software development process

- Steps in solving a problem:
  - Identify the features and plan the iterations
    - **Feature**
      - Defined as a client function
      - Specified as (action, result, object)
        - Action – a function that the application needs to provide
        - Result – obtained as a result of executing the function
        - Object – an entity where the application implements the function
      - *F1: finding the smallest prime number greater than given n*
    - **Iteration**
      - Time period when a stable and runnable version of a product is created (with documentation)
      - Helps to plan the delivery of features
      - ***I1=F1***

Software development process

- Steps in solving a problem:
  - List of activities or tasks (for each feature) and their description
    - Recommendations:
      - Define one activity for each operation
      - Define an activity for user interface (UI) interaction
      - Define an activity for UI operations
      - Determine the dependencies between activities
    - *A1: verify if a given number is prime or not*
    - *A2: find the smallest prime number greater than a given natural number*
    - *A3: implement the initialization of a number, finding the smallest prime number greater then n and return the result*
    - *A4: implement the UI*

Software development process

- Steps in solving a problem:
  - List of activities (for each feature) and their **description**
    - Testing cases
      - Specify a set of input data and expected results to evaluate a part of a program
      - *A1: verify if a given number is prime or not*

Input	Output
2	True
6	False
3	True
-2	False
1	False

Software development process

- Steps in solving a problem:
  - List of activities (for each feature) and their **description**
    - Implementation
      - *A1: verify if a given number is prime or not*

```
# Description: verifies if the number n is prime
# Data: n
# Precondition: n - natural number
# Results: res
# Postcondition: res=FALSE, if n is not prime or res=TRUE, if n is prime
if (n < 2):
    print("no ", n, " is not prime (is composed)")
else:
    d = 2
    isPrime = True
    while (d * d <= n) and (isPrime == True):
        if (n % d == 0):
            isPrime = False
        else:
            d = d + 1
    if (isPrime == True):
        print("no ", n, " is prime")
    else:
        print("no ", n, " is not prime")
```

Simple feature-driven development

- Build a feature list from problem statement
- Plan iterations
- For each iteration
  - Model planned features
  - Implement and test the features
  - Obs:
    - At the beginning of each iteration: analyze each feature – determine the activities (tasks) required – schedule the tasks – implement and test each independently.
    - *An iteration will result in a working program for the client (will interact with the user, perform some computation, show results)*

Programming paradigms

- Fundamental style of computer programming
- Imperative programming
  - Computations described through statements that modify the state of a program (control flow – sequence of statements executed by the computer)
  - Examples
    - **Procedural programming** – each program is formed by several procedures (subroutines or functions)
    - Object oriented programming
- Declarative programming
  - Expresses the logic of a computation (without describing the control flow)
  - Examples
    - Functional programming (LISP)
    - Logic programming (Prolog, SQL)

Procedural programming – functions

- **Procedural programming** – each program is formed by several procedures (subroutines or functions)
- Function
  - A block of statements that can be reused
  - Are run in a program only when they are called
  - Function characteristics:
    - Has a **name**
    - Has a list of **parameters**
    - Can **return** a value
    - Has a body (a block of statements)
    - Has a specification (**docstring**) formed of:
      - A description
      - Type and description of parameters
      - Conditions imposed on input parameters (pre-conditions)
      - Type and description of return value
      - Conditions imposed on output values (post-conditions)
      - Exceptions that can occur during its execution

Functions in Python

- A function is defined using reserved keyword def
- Execution of function is produced only upon calling / invoking it

```
def get_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

get_max(2, 3)
```

Functions in Python

- Calling a function
  - Recap. **block** = part of a Python program (identified by indentation) executed as a unit
  - The body of a function is a block
  - A block is executed in a new execution frame which:
    - Contains administrative information (useful in the debugging phase)
    - Determines where and how the execution of the program will continue (after the execution of the current block is completed)
    - Defines 2 name spaces (local and global) that affect the execution of the block

Functions in Python

- Calling a function
  - New **scope/frame/environment** created when enter a function
  - Name space
    - A container of names
    - Link between name and object
    - Features similar to a dictionary
  - Binding
    - Adding a name to the name space
  - Rebinding
    - Changing the link between a name and an object
  - Unbinding
    - Removing a name from the name space

Functions in Python

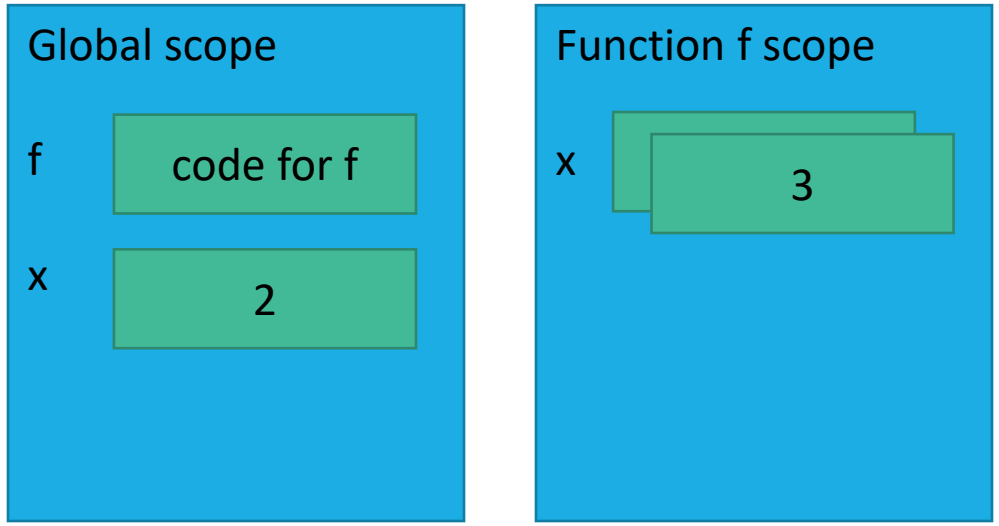
- Calling a function
  - Formal parameters
    - Identify input parameters
    - Each call to the function should respect the number and type of requested parameters
  - Actual parameters
    - Values given to the formal parameters when function is called
    - Stored in local symbol tables of the called function
    - Via reference

```
def search(element, lis):
    for x in lis:
        if (x == element):
            return True
    return False

a = [2, 3, 4, 5, 6]
el = 3
if (search(el, a) == True):
    print("el was found...")
else:
    print("el was not found...")
```

Variable scope

- Scope – defines if a variable is visible inside a block
  - Scope of a variable defined in a block is that block
  - Variables defined on a certain indentation level are considered local to that block

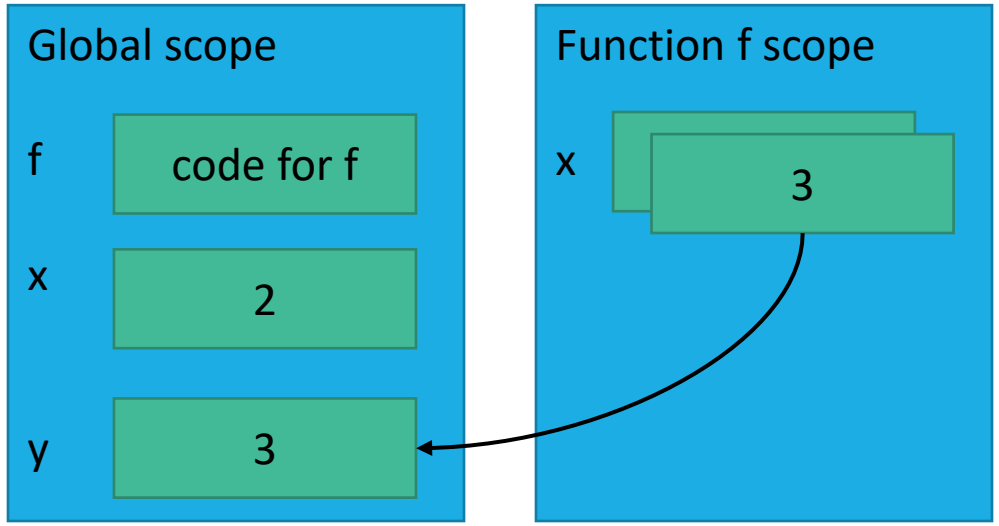
```
def f(x):
    x = x + 1
    print("Inside f, x = ", x)

x = 2
f(x)
```



Variable scope

- Scope – defines if a variable is visible inside a block
  - Scope of a variable defined in a block is that block
  - Variables defined on a certain indentation level are considered local to that block

```
def f(x):
    x = x + 1
    print("Inside f, x = ", x)
    return x

x = 2
y = f(x)
```



Variable scope

```
def f():
    a = 1
    a += 1
    print("Inside f, a = ", a)

a = 5
print("a = ", a)
f()
print("a = ", a)
```

a is redefined local variable

```
a =  5
Inside f, a =  2
a =  5
>>>
```

```
def f():
    print("Inside f, a = ", a)
    print("Inside f, a^2 = ", a ** 2)

a = 5
f()
print("a = ", a)
```

```
a =  5
Inside f, a =  5
Inside f, a^2 =  25
a =  5
>>>
```

```
def f():
    a += 1
    print("Inside f, a = ", a)

a = 5
print("a = ", a)
f()
print("a = ", a)
```

Unbound Local Error

```
a =  5
Traceback (most recent call last):
  File "C:\Users\cami\Desktop\c.py", line 7, in <module>
    f(a)
  File "C:\Users\cami\Desktop\c.py", line 2, in f
    a += 1
UnboundLocalError: local variable 'a' referenced before assignment
```

Variable scope

- Types of variables
  - **Local** – a name (of variable) defined in a block
  - **Global** – a name defined in a module
  - Free – a name used in a block but defined somewhere else

```
g1 = 1 # g1 - global variable (also local, a module being a block)

def fun1(a): # a is a formal parameter
    b = a + g1 # b - local variable, g1 - free variable
    if b > 0: # a, b, and g1 are visible in all blocks of this function
        c = b - g1 # b is visible here, also g1
        b = c # c is a local variable defined in this block
    return b # c is not visible here

def fun2():
    global g1
    d = g1 # g1 - global variable
    g1 = 2 # g1 must be declared global, before
    return d + g1 # any references to g1 in this function

print(fun1(1))
print(fun2())
```

Variable scope

- Where is a variable visible?
  - Rules to determine the scope of a name (variable or function)
    - A name is visible only inside the block where it is defined
    - The formal parameters of a function belong to the body of the function (are visible only inside the function)
    - Names defined outside of a function (at module level) belong to the module scope
    - When a name is used in a block, its visibility is determined using the nearest scope (that contains that name)

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

f()
print(a) # 100
```

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

f()
print(a) # 300
```

Variable scope

- Inspecting the local / global variables of a program
  - `locals()`
  - `globals()`

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

f()
print(a)
```

```
{'a': 500}
```

```
{'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__spec__': None, '__annotations__': {}, '__builtins__':
<module 'builtins' (built-in)>, '__file__':
'C:\\Users\\cami\\Desktop\\c.py', 'a': 300, 'f': <function f at
0x0065AC90>}
```

Variable scope

```
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))
```

```
Globals  {'__name__': '__main__', '__doc__': None, '__package__':
r__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': Non
ons__': {}, '__builtins__': <module 'builtins' (built-in)>, 'chang
able': <function change_or_not_immutable at 0x0055AC90>, 'g1': 1}
Before call: g1 =  1  id =  505571456
Locals  {'a': 1}
Before assignment: a =  1  id =  505571456
After assignment: a =  0  id =  505571440
After call: g1 =  1  id =  505571456
>>>
```

```
def change_or_not_mutable(a):
    print ('Locals ', locals())
    print ('Before assignment: a = ', a,' id = ', id(a))
    a[1] = 5
    print ('After first assignment: a = ', a,' id = ', id(a))
    a = [0]
    print ('After second 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))
```

```
Globals  {'__name__': '__main__', '__doc__': None, '_
r__': <class '_frozen_importlib.BuiltinImporter'>, '_
ons__': {}, '__builtins__': <module 'builtins' (built
rs\\cami\\Desktop\\c.py', 'change_or_not_mutable': <f
le at 0x027FAC90>, 'g2': [0, 1]}
Before call: g2 =  [0, 1]  id =  48222336
Locals  {'a': [0, 1]}
Before assignment: a =  [0, 1]  id =  48222336
After first assignment: a =  [0, 5]  id =  48222336
After second assignment: a =  [0]  id =  41938416
After call: g2 =  [0, 5]  id =  48222336
>>>
```

Recap today

- Simple feature-driven development
- Procedural programming
- Functions
  - Definition
  - Call
  - How to write a function
- Variable scope

Next time

- How to write functions - TDD
- Modular programming

Reading materials and useful links

1. The Python Programming Language - https://www.python.org/
2. The Python Standard Library - https://docs.python.org/3/library/index.html
3. The Python Tutorial - https://docs.python.org/3/tutorial/
4. M. Frentiu, H.F. Pop, Fundamentals of Programming, Cluj University Press, 2006.
5. MIT OpenCourseWare, Introduction to Computer Science and Programming in Python, https://ocw.mit.edu, 2016.
6. K. Beck, Test Driven Development: By Example. Addison-Wesley Longman, 2002. http://en.wikipedia.org/wiki/Test-driven_development
7. M. Fowler, Refactoring. Improving the Design of Existing Code, Addison-Wesley, 1999. http://refactoring.com/catalog/index.html