

## Lecture 3 – Program structure, Functions

<https://cw.fel.cvut.cz/wiki/courses/be5b33prg/start>

# Michal Reinštein

Czech Technical University in Prague,  
Faculty of Electrical Engineering, Dept. of Cybernetics,  
Center for Machine Perception

<http://cmp.felk.cvut.cz/~reinsmic/>  
[reinstein.michal@fel.cvut.cz](mailto:reinstein.michal@fel.cvut.cz)



# RECAP: LOOPS – FOR, WHILE



2

```
Python Console
/opt/local/bin/python3.6 /Applications/PyCharm.ap
Python 3.6.3 (default, Oct 5 2017, 23:34:28)
In[2]: for number in range(5):
.....:     print(number)
.....:
0
1
2
3
4
```

Special Variables

- `_ = {str} "`
- `__ = {str} "`
- `___ = {str} "`
- `number = {int} 4`

On each iteration or pass of the loop:

- Check to see if there are still more **items to be processed**
- If there are **none** left (the **terminating condition** of the loop) the loop has finished
- If there are items still to be processed, the **loop variable is updated** to refer to the next item in the list
- Program **continues at the next statement** after the loop body
- To explore: early **break**, or **for – else** loop, **while loop**

source [http://openbookproject.net/thinkcs/python/english3e/hello\\_little\\_turtles.html](http://openbookproject.net/thinkcs/python/english3e/hello_little_turtles.html)



```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n/x)
            break
```

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print( n, 'equals', x, '*', n/x)
            break
    else:
        # Loop fell through without finding a factor
        print(n, 'is a prime number')
```

- Recommendation: **early return / early break**
- Special condition: **FOR – ELSE**
- Explore on your own: **for, in, while, if, else, break, continue**



Global

Function definitions

Main section

```
example.py x
3  # import modules used here. sys is a very standard one
4  import sys
5
6
7  # Gather our code in a main() function
8  def main():
9      print('Hello there', sys.argv[1])
10     # Command line args are in sys.argv[1], sys.argv[2] ...
11     # sys.argv[0] is the script name itself and can be ignored
12
13
14 # Standard boilerplate to call the main() function to begin
15 # the program
16 if __name__ == '__main__':
17     main()
18
```

1. Global definitions section
2. Function definitions / classes definitions section
3. Sequence of instructions section (here the main section)



Global



Function definitions



Main section



```
example.py x
3 # import modules used here. sys is a very standard one
4 import sys
5
6
7 # Gather our code in a main() function
8 def main():
9     print('Hello there', sys.argv[1])
10     # Command line args are in sys.argv[1], sys.argv[2] ...
11     # sys.argv[0] is the script name itself and can be ignored
12
13
14 # Standard boilerplate to call the main() function to begin
15 # the program
16 if __name__ == '__main__':
17     main()
18
```

- When python interpreter runs a source file as main program, it sets **\_\_name\_\_** variable to have a value **"\_\_main\_\_"**
- If being imported from another module, **\_\_name\_\_** will be set to the **module's name**

source <https://developers.google.com/edu/python/introduction>

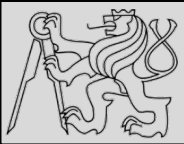


```
example.py x
3 # import modules used here -- sys is a very standard one
4 import sys
5
14 # Standard boilerplate to call the main() function to begin
15 # the program.
16 if __name__ == '__main__':
17     main()
18
```

- sys – access to exit(), argv, stdin, stdout, ...
  - re – regular expressions
  - os – operating system interface, file system
- You can find the documentation of all the Standard Library modules and packages at <http://docs.python.org/library>.

• Use **import** to include functions / classes from other modules

source <https://developers.google.com/edu/python/introduction>



# EXAMPLE

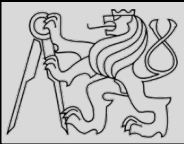


7

```
example.py x
1  #!/usr/bin/env python
2
3  # import modules used here -- sys is a very standard one
4  import sys
5
6
7  # Gather our code in a main() function
8  def main():
9      print('Hello there', sys.argv[0])
10     # Command line args are in sys.argv[1], sys.argv[2] ...
11     # sys.argv[0] is the script name itself and can be ignored
12     for n in range(2, 10):
13         print('n = ', n)
14         for x in range(2, n):
15             print('x = ', x)
16             if n % x == 0:
17                 print(n, 'equals', x, '*', n // x)
18                 break
19         else:
20             # loop fell through without finding a factor
21             print(n, 'is a prime number')
22
23
24 # Standard boilerplate to call the main() function to begin
25 # the program.
26 if __name__ == '__main__':
27     main()
28
```

Run example

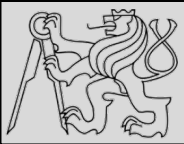
```
/opt/local/bin/python3.6 "/Users/michalreinstein/Di
Hello there /Users/michalreinstein/Di
n = 2
2 is a prime number
n = 3
x = 2
3 is a prime number
n = 4
x = 2
4 equals 2 * 2
n = 5
x = 2
x = 3
x = 4
5 is a prime number
n = 6
x = 2
6 equals 2 * 3
n = 7
x = 2
x = 3
x = 4
x = 5
x = 6
7 is a prime number
n = 8
x = 2
8 equals 2 * 4
n = 9
x = 2
x = 3
9 equals 3 * 3
Process finished with exit code 0
```



```
def NAME( PARAMETERS ):  
    STATEMENTS
```

- **Function** = named sequence of statements belonging together
- **Header line**: begins with a keyword **def**, ends with a colon **:**
- **Body**: one or more statements, each indented the same amount
- **Parameter list**: empty or any number of comma separated parameters (can have default value)
- Any **name** except for keywords and illegal identifiers
- Any **number of statements** inside the function, but **indented** from the **def** (standard indentation of **four spaces**)
- Function may or may not produce a result





- Organize program into **chunks** that match **how we think** about the problem
- Code **re-using** without copy-paste
- Enforcing logical **structure** into the code
- Easier **debugging**
- Code **readability**

source <http://openbookproject.net/thinkcs/python/english3e/functions.html>



**Import module math**

**Call sqrt() function**

**Use variable pi**

```
Python Console
/opt/local/bin/python3.6 /Applicat
Python 3.6.3 (default, Oct 5 2017)
>>> import math
>>> a = math.sqrt(9)
>>> radius = 3
>>> area = math.pi*radius**2
In[3]: print(area)
28.274333882308138
In[4]:
```

**Special Variables**

```
__ = {str} ""
__ = {str} ""
__ = {str} ""
a = {float} 3.0
area = {float} 28.274333882308138
radius = {int} 3
```

- <https://docs.python.org/3.4/library/math.html>



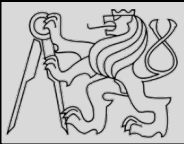
```
4 def compute_area_rectangle(height, width):
5     """
6     Compute area of rectangle
7     :param height: height of rectangle (m)
8     :type height: float
9     :param width: width of rectangle (m)
10    :type width: float
11    :return: area of rectangle (m^2)
12    :rtype: float
13    """
14    # use assert as function guard
15    assert height >= 0 and width >= 0, 'Length cannot be negative'
16    return height * width
```

Function guard



- Docstrings are meant for **documentation** (if the first thing after the function header is string then treated as docstring)
- Key way to **document** our functions
- Concept of abstraction (need to know the interface)
- Formed using **triple-quoted** strings
- Different from comments: retrievable by Python tools at **runtime** (comments are completely eliminated during parsing)

source <http://openbookproject.net/thinkcs/python/english3e/functions.html>



Swapping variables

```
Python Console
/opt/local/bin/python3.6 /Applications/PyCharm.a
Python 3.6.3 (default, Oct 5 2017, 23:34:28)
In[2]: x = 7
In[3]: y = 10
In[4]: x, y = y, x
In[5]: print(x)
10
In[6]: print(y)
7
```

- Flow of execution = **order of statements execution** (begins at the first statement of the program)
- Statements are executed **one at a time**, in order from top to bottom (but **read the flow**, not top to bottom!)
- Python evaluates **expressions from left to right** (during assignment right-hand side is evaluated first)
- Function calls are like a **detour** in the flow of execution
- We can define one function inside another
- Function **definitions do not alter the flow** of execution

source <http://docs.python.org/3/reference/expressions.html#evaluation-order>



```
1  #!/usr/bin/env python
2
3
4  def compute_area_rectangle(height, width):
5      # use assert as function guard
6      assert height >= 0 and width >= 0, 'Length cannot be negative'
7      return height * width
8
9
10 def compute_area_square(side_length):
11     return compute_area_rectangle(side_length, side_length)
12
13
14 ▶ if __name__ == '__main__':
15     square_side_length = float(input('Input square side length (m)'))
16     print(compute_area_square(square_side_length))
17
```

- Function **hide complex computation** behind a single command and capture abstraction of the problem.
- Functions can **simplify** a program
- Creating a new function can make a **program shorter** by eliminating **repetitive code**



# FUNCTIONS CALLING FUNCTIONS

```

1  #!/usr/bin/env python
2
3
4  def compute_area_rectangle(height, width):
5      """
6      ... Compute area of rectangle
7      ... :param height: height of rectangle (m)
8      ... :type height: float
9      ... :param width: width of rectangle (m)
10     ... :type width: float
11     ... :return: area of rectangle (m^2)
12     ... :rtype: float
13     """
14     ... # use assert as function guard
15     ... assert height >= 0 and width >= 0, 'Length cannot be negative'
16     ... return height * width
17
18
19  def compute_area_square(side):
20     ... return compute_area_recta
21
22
23  if __name__ == '__main__':
24     ... square_side_length = floa
25     ... print(compute_area_square
26

```

Function guard

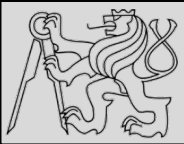
```

/opt/local/bin/python3.6 "/Users/michalreinstein/Disk
Google/TEACHING/BE5B33PRG_2017/examples/example_02.py"
Input square side length (m) -17
Traceback (most recent call last):
  File "/Users/michalreinstein/Disk Google/TEACHING/BE5B33PRG_2017/examples/example_02.py",
    line 25, in <module>
    ... print(compute_area_square(square_side_length))
  File "/Users/michalreinstein/Disk Google/TEACHING/BE5B33PRG_2017/examples/example_02.py",
    line 20, in compute_area_square
    ... return compute_area_rectangle(side_length, side_length)
  File "/Users/michalreinstein/Disk Google/TEACHING/BE5B33PRG_2017/examples/example_02.py",
    line 15, in compute_area_rectangle
    ... assert height >= 0 and width >= 0, 'Length cannot be negative'
AssertionError: Length cannot be negative

Process finished with exit code 1

```

source <http://openbookproject.net/thinkcs/python/english3e/functions.html>



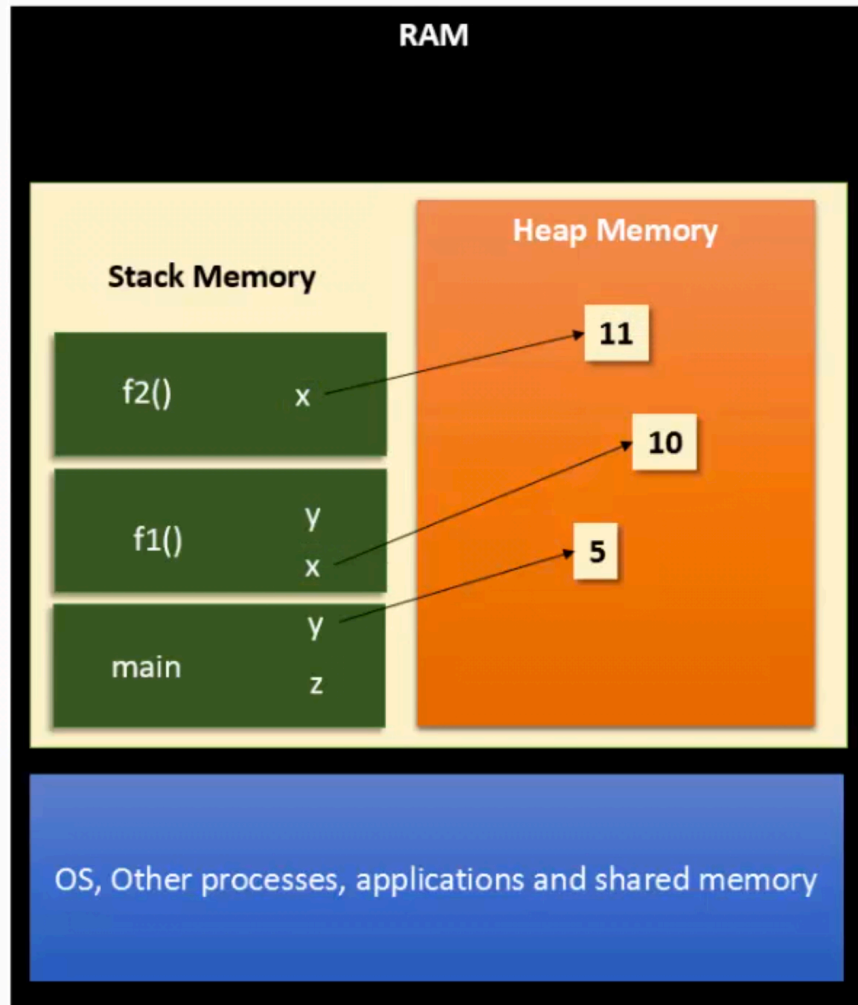
# FUNCTIONS CALLING FUNCTIONS



```

1  #!/usr/bin/env python
2
3
4  def f1(x):
5      x *= 2
6      y = f2(x)
7      return y
8
9
10 def f2(x):
11     x += 1
12     return x
13
14
15 if __name__ == '__main__':
16     y = 5
17     z = f1(y)
18     print(z)
19

```



```

4  def f1(x):
5      x *= 2
6      y = f2(x)
7      return y

```

Shadows name 'y' from outer scope [less...](#) (§F1)

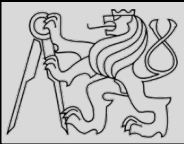
This inspection detects shadowing names defined in outer scopes.

```

11     x += 1
12     return x

```

source <https://www.youtube.com/watch?v=arxWaw-E8QQ&t=1s>



```
>>> abs(5)
5
>>> abs(-5)
5
```

```
>>> pow(2, 3)
8
>>> pow(7, 4)
2401
```

```
>>> max(7, 11)
11
>>> max(4, 1, 17, 2, 12)
17
>>> max(3 * 11, 5**3, 512 - 9, 1024**0)
503
```

- Most functions require **arguments** (named arguments, default values)
- More than one argument: e.g. **pow(base, exponent)**
- Functions like **range**, **int**, **abs** all return values that can be used to build more complex expressions
- Function that returns value is called a **fruitful function**
- Opposite of a fruitful function is **void function** (procedure)





```
1 def final_amt(p, r, n, t):
2     a = p * (1 + r/n) ** (n*t)
3     return a
```

If we try to use `a`, outside the function, we'll get an error:

```
>>> a
NameError: name 'a' is not defined
```

- When a variable is created inside a function, it is **local** and cannot be used outside (**shadowing names**)
- The variable `a` is local to `final_amt`
- Local variables only exist while the function is being executed — this is called variable **lifetime**
- Parameters are local and act like local variables



## Temporary variable

```
1 def area(radius):  
2     b = 3.14159 * radius**2  
3     return b
```

```
1 def area(radius):  
2     return 3.14159 * radius * radius
```

- Functions such as **abs**, **pow**, **int**, **max**, **range**, produce results
- Return statement of fruitful functions includes a **return value**
- Temporary variables like **b** above make **debugging** easier



```
1 def absolute_value(x):
2     if x < 0:
3         return -x
4     else:
5         return x
```

```
1 def absolute_value(x):
2     if x < 0:
3         return -x
4     return x
```

```
1 def bad_absolute_value(x):
2     if x < 0:
3         return -x
4     elif x > 0:
5         return x
```

```
>>> print(bad_absolute_value(0))
None
```

- Multiple return statements, one in each branch of conditional
- Code after return is called **dead code**, or unreachable code
- All Python functions return **None** whenever they do not return another value.



```
1 def is_divisible(x, y):  
2     """ Test if x is exactly divisible by y """  
3     if x % y == 0:  
4         return True  
5     else:  
6         return False
```

```
1 def is_divisible(x, y):  
2     return x % y == 0
```

Boolean functions are often used in conditional statements:

```
1 if is_divisible(x, y):  
2     ... # Do something ...  
3 else:  
4     ... # Do something else ...
```

- Functions that return Boolean values
- Give Boolean functions names that sound like yes/no questions, e.g. **is\_divisible**
- Condition of the **if** statement is itself a **Boolean expression**



```
example_02.py x example_03.py x example_04.py x Run example_04
1 def find_first_2_letter_word(xs):
2     for wd in xs:
3         if len(wd) == 2:
4             return wd
5     return "Nothing found"
6
7
8 print(find_first_2_letter_word(["This", "is", "a", "dead", "parrot"]))
9 print(find_first_2_letter_word(["I", "like", "cheese"]))
10
```

```
/opt/local/bin/python3.6 "/Users/mic
is
Nothing found

Process finished with exit code 0
```

- Return statement in the middle of a **for** loop – control **immediately returns** from the function
- EXAMPLE: *Let us assume that we want a function which looks through a list of words. It should return the first 2-letter word. If there is not one, it should return “Nothing found”*



$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- **Incremental development** technique – avoid long debugging sessions by adding and testing only a small amount of code at a time.
- EXAMPLE: *We want to find the distance between two points, given by the coordinates  $(x_1, y_1)$  and  $(x_2, y_2)$ . (Pythagorean theorem)*

*What are the inputs (parameters)?  
What is the output (return value)?*



## Define interface



```
1 def distance(x1, y1, x2, y2):  
2     return 0.0
```

```
>>> distance(1, 2, 4, 6)  
0.0
```

## Process parameters



```
1 def distance(x1, y1, x2, y2):  
2     dx = x2 - x1  
3     dy = y2 - y1  
4     return 0.0
```

```
>>> distance(1, 2, 4, 6)  
0.0
```

## Temporary variables



```
1 def distance(x1, y1, x2, y2):  
2     dx = x2 - x1  
3     dy = y2 - y1  
4     dsquared = dx*dx + dy*dy  
5     return 0.0
```

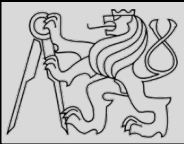
```
>>> distance(1, 2, 4, 6)  
0.0
```

## Return result



```
1 def distance(x1, y1, x2, y2):  
2     dx = x2 - x1  
3     dy = y2 - y1  
4     dsquared = dx*dx + dy*dy  
5     result = dsquared**0.5  
6     return result
```

```
>>> distance(1, 2, 4, 6)  
5.0
```

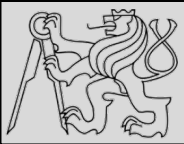


```
1 import math
2
3 def distance(x1, y1, x2, y2):
4     return math.sqrt( (x2-x1)**2 + (y2-y1)**2 )
```

```
>>> distance(1, 2, 4, 6)
5.0
```

- Start with a working **skeleton program** and make small **incremental changes** (analyze errors)
- Use **temporary variables** to refer to intermediate values for easy inspection
- Once the program is working, **explore options** and parameters
- Consolidate multiple statements to make **shorter code**, **refactor for readability**





These are the terms you should explore and know:

- **Argument**
- **Header**
- **Body**
- **Docstring**
- **Flow of execution**
- **Frame**
- **Function**
- **Function call**
- **Function composition**
- **Function definition**

- **Fruitful function**
- **Header line**
- **Import statement**
- **Lifetime**
- **Local variable**
- **Parameter**
- **Refactor**
- **Stack diagram**
- **Traceback (stack trace)**
- **void function**

Learning with Python 3 - chapter 4.8

<http://openbookproject.net/thinkcs/python/english3e/functions.html>

source <http://openbookproject.net/thinkcs/python/english3e/functions.html>



The formula for computing the final amount if one is earning compound interest is given on Wikipedia as

$$A = P \left( 1 + \frac{r}{n} \right)^{nt}$$

Where,

- $P$  = principal amount (initial investment)
- $r$  = annual nominal interest rate (as a decimal)
- $n$  = number of times the interest is compounded per year
- $t$  = number of years

Write a Python program that assigns the principal amount of \$10000 to variable  $P$ , assign to  $n$  the value 12, and assign to  $r$  the interest rate of 8%. Then have the program prompt the user for the number of years  $t$  that the money will be compounded for. Calculate and print the final amount after  $t$  years.



$$A = P \left( 1 + \frac{r}{n} \right)^{nt}$$

Where,

- P = principal amount (initial investment)
- r = annual nominal interest rate (as a decimal)
- n = number of times the interest is compounded per year
- t = number of years

```
1 def final_amt(p, r, n, t):
2     """
3     Apply the compound interest formula to p
4     to produce the final amount.
5     """
6
7     a = p * (1 + r/n) ** (n*t)
8     return a          # This is new, and makes the function fruitful.
9
10 # now that we have the function above, let us call it.
11 toInvest = float(input("How much do you want to invest?"))
12 fnl = final_amt(toInvest, 0.08, 12, 5)
13 print("At the end of the period you'll have", fnl)
```

- Will be evaluated and returned to the caller as the “**fruit**”
- Input **prompt** from user (**type conversion** from string to float)
- Arguments for **8%** interest, compounded **12** times per year, for **5** years period
- NOTE: It is as if **p = toInvest** is executed when **final\_amt** is called (variable name in the caller does not matter, in **final\_amt** the name is **p** with **lifetime** until return)



# EXAMPLE



$$A = P \left( 1 + \frac{r}{n} \right)^{nt}$$

Where,

- P = principal amount (initial investment)
- r = annual nominal interest rate (as a decimal)
- n = number of times the interest is compounded per year
- t = number of years

```
1 def final_amt(p, r, n, t):
2     """
3     Apply the compound interest formula to p
4     to produce the final amount.
5     """
6
7     a = p * (1 + r/n) ** (n*t)
8     return a          # This is new, and makes the function fruitful.
9
10 # now that we have the function above, let us call it.
11 toInvest = float(input("How much do you want to invest?"))
12 fnl = final_amt(toInvest, 0.08, 12, 5)
13 print("At the end of the period you'll have", fnl)
```

```
1 def final_amt_v2(principalAmount, nominalPercentageRate,
2                  numTimesPerYear, years):
3     a = principalAmount * (1 + nominalPercentageRate /
4                            numTimesPerYear) ** (numTimesPerYear*years)
5     return a
6
7 def final_amt_v3(amt, rate, compounded, years):
8     a = amt * (1 + rate/composed) ** (composed*years)
9     return a
```