Writing Functions

1DV501/1DT901: Introduction to programming

Jonas Lundberg, office B3024

Jonas.Lundberg@lnu.se

The slides are available in Moodle

September 9, 2022

Today ...

- **Writing your own functions**
- **Parameter passing**
- Global variables
- Default parameters
- **Organizing one file programs**
- **A separate file with only functions**
- Extra material
  - Documenting functions
  - Functions as parameters

boldface $\Rightarrow$ important!

Reading instructions: 7.1-7.3, 8.1-8.2
The important parts are 7.1-7.2, 8.1-8.2

A first function example

```
# Function definition
def increment(n):
    p = n + 1          # Function body
    return p


# Program starts
x = 1
y = increment(x)  # Call function increment
print(x, y)    # Output: 1 2

p = 7
q = increment(p)  # Call function increment
print(p, q)    # Output: 7 8
```

- The code `def increment(...) ...` defines a new function named `increment`
- We later **call** this function as `q = increment(p)`
- A function must be defined before they are used
  $\Rightarrow$ above the code that is using it
- Execution starts in the program and jumps temporarily to `increment` each time it is called.

A function with no return values

```
# Function definition with no return
def print_countdown(n):
    if n < 1:
        print("It must be a positive number!")
    else:
        for i in range(n,0,-1):
            print(i, end=" ")
        print()    # line break


# Program starts
print_countdown(10)      # Output: 10 9 8 7 6 5 4 3 2 1
print_countdown(-1)      # Output: It must be a positive number!
```

- The variable `n` in `print_countdown(n)` is called a **parameter**
- The values used in the calls (10 and -1) are called **arguments**
- A function can have zero or more return values. The example above has zero return values.

Multiple parameters and return values

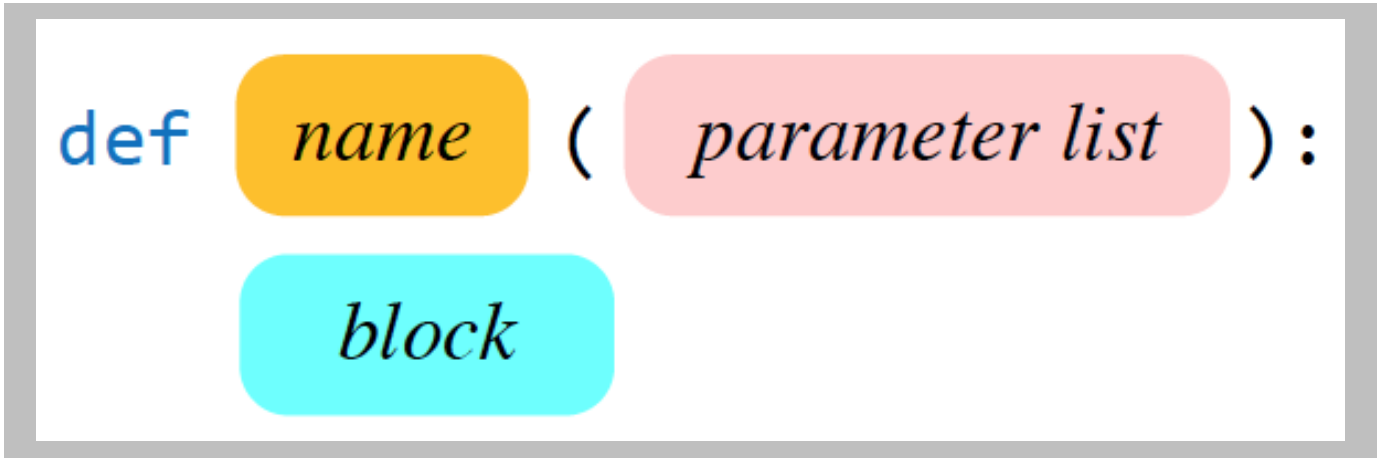
```
# Function with multiple parameter values
def add_all(a, b, c):
    return a+b+c

# Function with multiple return values
def increase_decrease(n):
    a = n + 1
    b = n - 1
    return a, b      # Return two values


# Program starts
x = add_all(1,2,3)                  # x = 6
p, q = increase_decrease(x)         # Take care of two return values
print(p, q)                         # Output: 7 5
```

- Function `add_all` has three parameters and one return value
- Function `increase_decrease` has one parameter and two return values
- Returning two values $\Rightarrow$ handle the return values using a multi-assignment
  `p, q = increase_decrease(x)`

Functions - Rules



- The `def` keyword marks the beginning of the function's definition
- Each function has a name that we later on use to call it
- A function may have zero or more parameters
- A function with N parameters requires N arguments when called
- The function body (`block` in figure above) makes use of the parameters to compute and return zero or more results
- Keyword `return` ⇒ function execution stops (and returns to the call site)
- Functions must be defined before (in the code) they are called
- Two functions in one file can not have the same name ⇒ no function overriding

More Examples

```
# A number n>0 is prime if not dividable by any number in range [2,n-1]
def is_prime(n):
    if n < 2:
        return False
    else:
        for i in range(2, n):
            if n % i == 0:
                return False
        return True

# Program starts
p = 25
print("is_prime:", is_prime(p))    # Output is_prime: False
```

- We have multiple (3) return statements
- `return` ⇒ execution immediately stopped and value returned
- `if n % i == 0: return False` ⇒ we interrupt loop once we know the result

Examples with Short Alternatives

```
# Is a larger than b?
def larger_than(a, b):
    if a > b:
        return True
    else:
        return False
    # return a > b    # Short alternative

# Return longest of two strings
def get_longest(s1, s2):
    if len(s1) > len(s2):
        return s1
    else:
        return s2
    # return s1 if len(s1) > len(s2) else s2       # Short alternative

# Program starts
a, b = 5, 7
print("larger_than:", larger_than(a, b))            # False
fn, sn = "Jonas", "Lundberg"
print("get_longest:", get_longest(fn, sn))          # Lundberg
```

Functions - Best practice

- Functions are named in the same way as variables. That is, they start with a lower case letter and words are separated by an underscore.
- **Try to make your functions reusable**. They should do one thing, and they should do it in a good way. Example: The function `sort_and_print(...)` should probably be split into functions `sort(...)` and `print(...)` since it is much more likely that each one of the shorter functions can be reused later on
- When to use functions?
  - When your program starts to get too long $\Rightarrow$ divide it into smaller parts $\Rightarrow$ divide into functions
  - When you repeat the same type of computations many times $\Rightarrow$ make a function of the computation and call it many times.
    Advantages: Shorter code and easier to update function (than multiple occurrences of similar code) when error in computation discovered.
  - Functions are name given computations $\Rightarrow$ makes program easier to understand. For example, consider a function `is_prime_number(n)`, the name says that we check if a given number is a prime number.

Parameter passing and local variables

```
def add_one(n):
    n = n + 1

# Program starts
a = 10
add_one(a)
print(a)

n = 5
add_one(n)
print(n)
```

Q: What is printed in the two cases? A: 10 and 5 are printed

- Parameter `n` in function `add_one` is a local variable ⇒ not same `n` as in the program below
- At the call `add_one(n)`, parameter `n` inside `add_one` is assigned the value 5 and updates it. However, the update has no effect on the program since `n` is not the same variable as the program variables `n` and `a`.
- Parameters and variables defined inside a function are **local** to that function ⇒ they are not the same parameters/variables that are used in other functions or in the main program.

Local variables

- Parameters and variables defined inside a function are **local** to that function
  $\Rightarrow$ they are not the same parameters/variables that are used in other functions or in the main program
  $\Rightarrow$ the same parameter/variable name can be used in different functions without any conflict.
- The memory required to store a local variable is used only when the variable function is executed. When the program's execution leaves the function, the memory for that variable is freed up.

Property 1 is very import for practical reasons, property 2 is only import for very large programs (or in programs with very many function calls).

Flake8 recommendations

```
def add_all(a, b, c):
    return a+b+c


def increase_decrease(n):
    a = n + 1
    b = n - 1
    return a, b      # Return two values


# Program starts
x = add_all(1, 2, 3)            # x = 6
p, q = increase_decrease(x)     # Take care of two return values
print(p, q)                     # Output: 7 5
```

Flake 8 requires

- A space after each comma in parameter list (e.g. `add_all(a, b, c)`)
- A space after each comma in call argument list (e.g. `add_all(1, 2, 3)`)
- Two empty lines after each function definition

Programming example: Many functions

Problem Inside file `simple_functions.py` create the functions we describe below and program code showing how the different functions can be used

- A function `is_odd(n)` that returns True if the integer n is odd, otherwise False.
- A function `sum_range(m, n)` that returns the sum off all integers in the range `[m, n]` (both m and n included). You can assume that both n and m are non-negative and that $n > m$.
- A function `contains(s, c)` that returns True if string s contains character c, otherwise False.
- A function `multi_print(s, n)` that prints the string s n times in a single white space separated line.
- A function `hasXandY(str)` returning True if the input string str contains both the upper case letters X and Y (and False otherwise). That is, the strings `abbX`, `aYbx`, and `YYYY` should all return False whereas `YbbX`, `XXYYXX`, and `XYlofon` should all return True.

Notice: You can assume that the arguments used in a call to any of these functions are of the correct (expected) type. Also, notice that this exercise doesn't require any input.

Solution: is_odd, sum_range, multi_print

```
# True if n is odd, otherwise False
def is_odd(n):
    if n % 2 == 1:
        return True
    else:
        return False
    # return n % 2 == 1    # Short alternative solution

# Returns sum of all integers in range [m,n]
def sum_range(m, n):
    sum = 0
    for i in range(m, n+1):
        sum = sum + i
    return sum

# Print string s n times in a single whitespace separated line
def multi_print(s, n):
    for i in range(n):
        print(s, end=" ")
    print()     # Break line
```

Solution: contains, hasXandY(s)

```
# True if s contains character c, otherwise False
def contains(s, c):
    for ch in s:
        if ch == c:
            return True
    return False
    # return c in s        # Short alternative solution

# True if s contains both X and Y, otherwise False
def has_XandY(s):
    x, y = False, False
    for c in s:         # For each character in string
        if c == 'X':
            x = True
        elif c == 'Y':
            y = True
    return x and y        # Both must be true
    # return "X" in s and "Y" in s  # Short alternative solution
```

Notice: `"c in s"` is `True` if string `s` contains character `c`

Solution: Test all functions

```
# Program starts
n = 7
print("is_odd:", is_odd(n))     # True

m, n = 5, 10
print("sum_range:", sum_range(m, n))     # 45

s = "Jonas"
c = "j"
print("contains:", contains(s, c))       # False

s = "Hello"
n = 5
print("multi_print:", end=" ")
multi_print(s, n)     # No return value expected

s = "XylofonY"
print("hasXandY:", hasXandY(s))     # True
```

While implementing, add one function followed by their test code. Try to verify that a functions works correctly as soon as possible.

A 10 minute break?

zzzzzzzzzzzzzzz ...

Global variables (1)

```
# Introduce two global variables
n1 = 0
n2 = 0

def get_input():
    global n1, n2
    n1 = int( input("Enter integer 1: ") )  # Update global n1
    n2 = int( input("Enter integer 2: ") )

# Program starts
get_input()   # Assigns new values to n1 and n2
print(f"Integer 1 is {n1} and Integer 2 is {n2}")  # Use globals n1,n2
```

Execution example:

```
Enter integer 1: 7
Enter integer 2: 9
Integer 1 is 7 and Integer 2 is 9
```

- Variables defined before any functions are **global variables**
- Global variables can be accessed in all functions and in the main program
- **Warning:** Global variables makes program hard to read ⇒ try to avoid them

Global variables (2)

```
n = 0  # Global variable n

def set_global_1(a):
    global n
    n = a   # Updates global variable n
def set_global_2(a):
    n = a   # Updates local variable n
def get_global():
    return n   # Returns global n, no declaration needed

set_global_1(5)
print(n)       # Print global n, output: 5

set_global_2(7)
print(n)       # Print global n, output is still 5

print( get_global() )  # Print current global value ==> Output: 5
```

- To update a global variable inside a function you need to declare it as `global`
- Not declared as global ⇒ considered as introducing a new local variable
- No need to declare global when only reading a global variable

Organizing single file programs

Recommended file organization

Simplest possible

1. Import statements
2. Global variables
3. Function definitions
4. Program starts

Approach used so far

Using a main function

1. Import statements
2. Global variables
3. Function definitions
4. A function main() containing the program
5. A call to main() to start program

Approach sometimes used in textbook by Halterman

Motivation for using `main()`: Functions help to organize our code.

The name `main` for the controlling function is arbitrary but traditional; several other popular programming languages (C, C++, Java, C, Objective-C) require such a function and require it to be named main.

The `main()` function approach

```
def increase(n):
    return n + 1

def decrease(n):
    return n - 1

def main():       # Function representing program
    p = 7
    p = increase(p)
    p = increase(p)

    q = 7
    q = decrease(q)

    print(p, q)     # Output: 9 6

main()   # Call main to start program
```

Feel free to use the `main()` function approach. Personally I (Jonas) think we can do without it as long as we clearly signal with comments where the program starts.

Default Parameters (1)

Python allows us to give certain parameters a default value
⇒ values to be used if parameter not used

```
# Prints all integers in range [n,m] on a single line
def print_range(n = 0, m = 5):
    for i in range(n, m + 1):
        print(i, end=" ")
    print()

# Program starts
print_range()        # Use default values ==> 0 1 2 3 4 5
print_range(6, 10)   # Non-default values ==> 6 7 8 9 10
print_range(3)       # n = 3, m = 5 ==>  3 4 5
```

- The function parameters default values are `n = 0, m = 5`
- Call `print_range()` ⇒ both default values are used
- Call `print_range(6, 10)` ⇒ overrides default values ⇒ defaults are not used
- Call `print_range(3)` ⇒ overrides 1st default, 2nd default values is used

Default Parameters (2)

```
def print_range(n, m = 5):     # Only 2nd parameter has default value - OK!
    for i in range(n, m + 1):
        print(i, end=" ")
    print()

def print_range(n = 0, m):     # Only 1st parameter has default value - Error!
    for i in range(n, m + 1):
        print(i, end=" ")
    print()
```

- A parameter with a default value is called a **default parameter**
- A function can have any number of default parameters
- However, the default parameters must come in the end of the parameter list
- A default parameter (`n = 0`) can not be followed by non-default parameter (`m`)

Using multiple .py-files

Dividing your program into several files is simple

My library (or module) file `B.py`

```
def increase(n):
    return n + 1

def decrease(n):
    return n - 1
```

- Simple library (or module) ⇒ a collection of functions
- Functions can be re-used in many programs
- **The library file must be in the same folder as the program** for this simple approach to work
- A necessary approach when your program gets larger

Using functions in `B.py` (Version 1)

```
import B   # Make all functions in B available

p = 7
p = B.increase(p)  # B must be referenced
p = B.increase(p)
p = B.decrease(p)
print(p)                  # Output: 8
```

Using functions in `B.py` (Version 2)

```
# Make only increase and decrease available
from B import increase, decrease

p = 7
p = increase(p)   # No need to reference B
p = increase(p)
p = decrease(p)
print(p)                   # Output: 8
```

Programming Example: Use multiple files

Exercise: Split the previous program `simple_functions.py` into two files:

- File `my_functions.py` containing all the functions (`is_odd`, ...)
- File `my_main.py` containing the test code in `simple_functions.py`.

Solution idea

- Part 1 is trivial. Just create `my_functions.py` and move (copy and paste) all functions from `simple_functions.py` to `my_functions.py`
- Part 2: Create `my_main.py`, copy and paste test code from `simple_functions.py`, add import statement and update function calls. See next slide.

Notice: This simple approach only works for files in the same folder. Accessing modules in other folders is more difficult.

Solution: my_main.py

```
import my_functions as mf  # Give it a short name "mf"

# Program starts
n = 7
print("is_odd:", mf.is_odd(n))     # Use "mf" in each call
                                   # to external functions

m, n = 5, 10
print("sum_range:", mf.sum_range(m, n))

s = "Jonas"
c = "j"
print("contains:", mf.contains(s, c))

s = "Hello"
n = 5
print("multi_print:", end=" ")
mf.multi_print(s, n)     # No return value expected

s = "XylofonY"
print("hasXandY:", mf.hasXandY(s))
```

Function documentation

```
def gcd(a, b):
    """The Euclidean algorithm for computing the greatest
       common divisor of integers a and b. First presented 300 BC."""
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a

# Program starts
p = gcd(60,45)
print(p)   # Output: 15
```

- The recommended approach to document a function in Python is inside """ ... """ (triple quotes) in the beginning of the function body.
- Software tools can extract this information and generate code documentation
- One usually document a) the purpose of the function, b) The role of each parameter (value type and what it means), c) the return value (value types and what it means), d) a reference (if idea taken from someone else)
- **Not used in this course. But you should recognize them.**

Functions as values

```
from math import sqrt

x = sqrt                        # Assign function sqrt to variable
print(x(16), type(x))           # Apply function sqrt using variable x

sqrt = 7                        # Redefine sqrt ==> sqrt no ...
print(sqrt, type(sqrt))         # longer a function (in this program)

print = 7                  # Redefine print
print("hello")             # Error, print function no longer available
```

Output:

```
4.0 <class 'builtin_function_or_method'>
7 <class 'int'>
TypeError: 'int' object is not callable
```

- Functions are also a type of values in Python. They can be assigned to variables and used as parameters in calls.
- Function names can be redefined ⇒ they lose the original functionality (Be careful, redefining function names is usually a bad idea.)

Functions as parameters

```
def plus(a, b):
    return a + b

def minus(a, b):
    return a - b

def apply_op(a, b, op):    # Expects two numbers and a function
    return op(a,b)         # with two parameters as input

# Program starts
p = apply_op(6, 3, plus)  # Use plus(a,b) as argument
q = apply_op(6, 3, minus) # Use minus(a,b) as argument
print(p, q)               # Output: 9 3
```

- The function `apply_op(a, b, op)` expects two numbers and a function with two parameters as input
- We call it by providing a two-parameter function (like `plus`) as an argument
- An "advanced" concept" that will be used later on, not part of Assignment 2