Writing Functions

1DV501/1DT901: Introduction to programming

Jonas Lundberg, office B3024

Jonas.Lundberg@lnu.se

The slides are available in Moodle

September 16, 2020

Today ...

- **Writing your own functions**
- Parameter passing
- Global variables
- Default parameters
- Organizing one file programs
- A separate file with only functions
- Recursion (Introduction)
- If time permits
  - Documenting functions
  - Functions as parameters

Reading instructions: 7.1-7.3, 8.1-8.5
The important parts are 7.1-7.2, 8.1-8.4

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 the function definition is called a **parameter**
- The values used the call (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

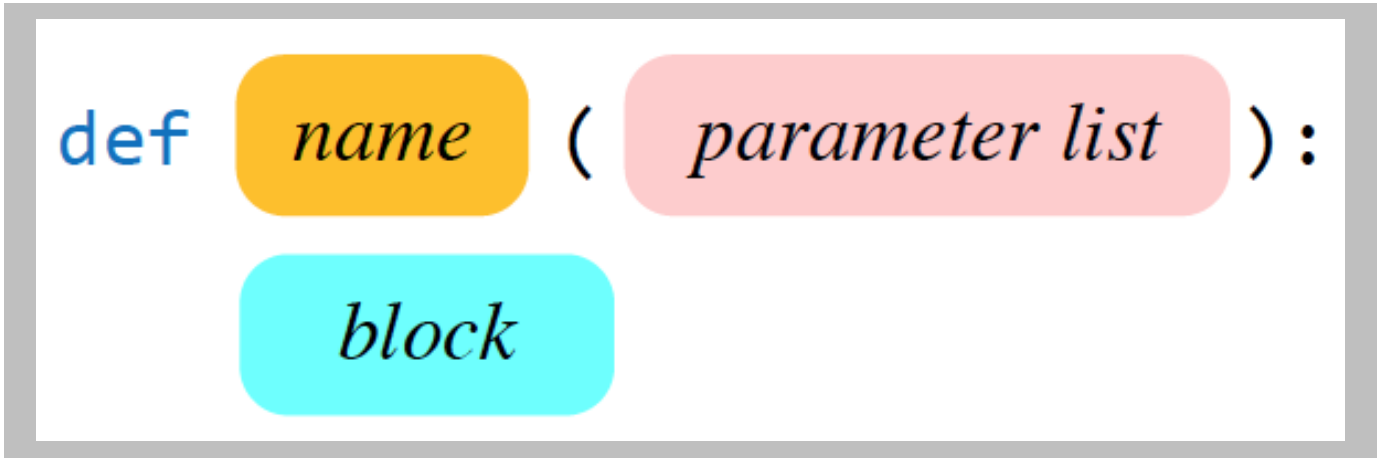
```
# Function with multiple parameter values
def add_all(a, b, c):
    return a+b+c

# 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 functions 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

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 methods 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).

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 {n1}")  # Use global n1
```

Execution example:

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

- 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 directory 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: `has_XandY(str)`

Problem

- Inside a file `xandy.py`, write a function `has_XandY(str)` returning True if the input string str contains both the upper case letters X and X (and False otherwise). That is, the strings `abbX`, `aYbx`, and `YYYY` should all return False whereas `YbbX`, `XXYYXX`, and `XYlofon` should all return True.
- Also, inside file `xandy.py`, present a short program that demonstrates how the function can be used.

Solution: has_XandY(str)

```
def has_XandY(str):
    x, y = False, False
    for c in str:          # For each character in string
        if c == 'X':
            x = True
        elif c == 'Y':
            y = True
    return x and y         # Both must be true

def test_and_print(s):
    if has_XandY(s):
        print(s, "contains both X and Y")
    else:
        print(s, "doesn't contain both X and Y")

# Program starts
test_and_print("XYlofon")              # True
test_and_print("Xylofon")              # False
test_and_print("aXyYb")                # True
test_and_print("aXyb")                 # False
```

A 10 minute break?

zzzzzzzzzzzzzzz ...

Recursion - An introduction

Recursion: A solution to a problem based on a smaller (but similar) problem.

Example: The sum of all integers in range 1 to n

$$S(n) = \sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-2) + (n-1) + n$$

- The sum can be computed using iteration:

```
# Computes the sum 1+2+3+...n for any n > 0
def sum(n):
    s = 0
    for i in range(1,n+1):
        s = s + i
    return s

# Program starts
p = sum(100)
print("The sum 1+2+3+4+...+100 is ", p)    # Output: 5050
```

Computing sum using smaller sums

$$S(n) = \sum_{i=1}^{n} i = \underbrace{1 + 2 + 3 + \cdots + (n-2) + (n-1)}_{S(n-1)} + n$$

- The problem can be expressed using a smaller problem: $S(n) = S(n-1) + n$
- **Ex**: S(5) = S(4) + 5
- And moving on ...
  - S(4) = S(3) + 4
  - S(3) = S(2) + 3
  - S(2) = S(1) + 2
  - S(1) = S(0) + 1
  - S(0) = S(-1) + 0 ???

We must find a base case ⇒ a case where it all stops!

Arithmetic Sum: Introducing a Base Case

- We need a *base case* to terminate the computation.
- We choose to set the base case to $S(1) = 1$ ($S(0) = 0$ would also work).
- The base case is expressed as a fact, not referring to any smaller problems.
- We now have a **recursive definition**:

$$S(n) = \begin{cases} 1 & n = 1 \quad (\textit{base case}) \\ S(n-1) + n & n \geq 2 \quad (\textit{recursive step}) \end{cases}$$

- As a recursive Python function

```
# Recursive computation 1+2+3+...n for any n > 0
def sum_rec(n):
    if n == 1:
        return 1
    else:
        return n + sum_rec(n-1)      # A recursive call
```

Recursion in practice ⇒ a function calls itself

Executing recursive sum

```
# Recursive computation 1+2+3+...n for any n > 0
def sum_rec(n):
    if n == 1:
        return 1
    else:
        return n + sum_rec(n-1)
```

Executing `sum_rec(5)` $\Rightarrow$ 5 calls to `sum_rec(...)`

```
sum_rec(5)
   sum_rec(4)
      sum_rec(3)
         sum_rec(2)
            sum_rec(1)
            return 1                    // base case
         return 2 + 1                 (= 3)
      return 3 + 3                  (= 6)
   return 4 + 6                  (= 10)
return 5 + 10                 (= 15)
```

Recursion

- Compute a solution to a problem using a smaller (but similar) problem is called *recursion*.
- In general, recursion $\Rightarrow$ a method calls itself.
- In order not to be trapped in an inifinite loop, a base case (at least one) must be part of the definition.
- Everything that can be done recursively, can also be done iteratively but not always as easy.
- Recursive definitions and algorithms are common in mathematics and computer science.
- Well-known problems where recursion helps: Fibonacci numbers, Binary search trees

Simple palindrome: A recursive definition

- A string is a *simple palindrome* if it has the same text in reverse.
- Examples: x, anna, madam, abcdefedcba, yyyyyyyy
- A palindrome can be defined as:
  1. An empty string is a palindrome
  2. A string with the length 1 is a palindrome.
  3. A string is a palindrome if the first and last characters are equal, and all characters in between is a palindrome.
- 1 and 2 are our base cases
- 3 is our recursive step

Simple palindrome: Python

```
# str is text to be checked,
# p and q are first and last positions in text
def is_palindrome_rec(str, p, q):
    if q <= p:
        return True
    elif str[p] != str[q]:
        return False
    else:
        return is_palindrome_rec(str, p+1, q-1)

# Programs starts
s = "madam"
if is_palindrome_rec(s,0,4):
    print(s, " is a simple palindrome")
else:
    print(s, " is not a simple palindrome")
```

Notice: We must not only call `is_palindrome_rec` with the text to be checked, we must also provide the first and last positions in the text

Recursive helper functions

Avoid having to provide (the rather ugly) first and last positions in the text to be checked using a recursive help function

```
def is_palindrome_rec(str, p, q):
    "... see previous slide ..."

# Help function that initializes the recursive function
def is_palindrome(str):
    p = 0                 # First position
    q = len(str) - 1      # Last position
    return is_palindrome_rec(str, p, q)

# Programs starts
s = "jonas"
if is_palindrome(s):      # A better looking call
    print(s, " is a simple palindrome")
else:
    print(s, " is not a simple palindrome")
```

Hence, by introducing a help function we can avoid providing first and last positions in the text to be checked $\Rightarrow$ a better looking function `is_palindrome`

Example: The Fibonacci Sequence

- In the *Fibonacci* sequence the first two numbers are 0 and 1 and the others are the sum of the two previous numbers.

  `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...`

- **Exercise**: Write a recursive function `fib(n)` computing the *n*:th number in the Fibonacci sequence. For example `fib(0) = 0`, `fib(1) = 1`, and `fib(6) = 8`.

- **Solution**:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

# Program starts
print( fib(6) )    # Output: 8
print( fib(15) )   # Output: 610
```

The First 50 Fibonacci Numbers

- Problem: Print the first 50 numbers in the Fibonacci sequence.
- **Solution:** Simple!

```
for i in range(0, 51):
    print(i, fib(i) )
```

- **Result:** The printing goes slower and slower and then dies.
- **A better solution:**

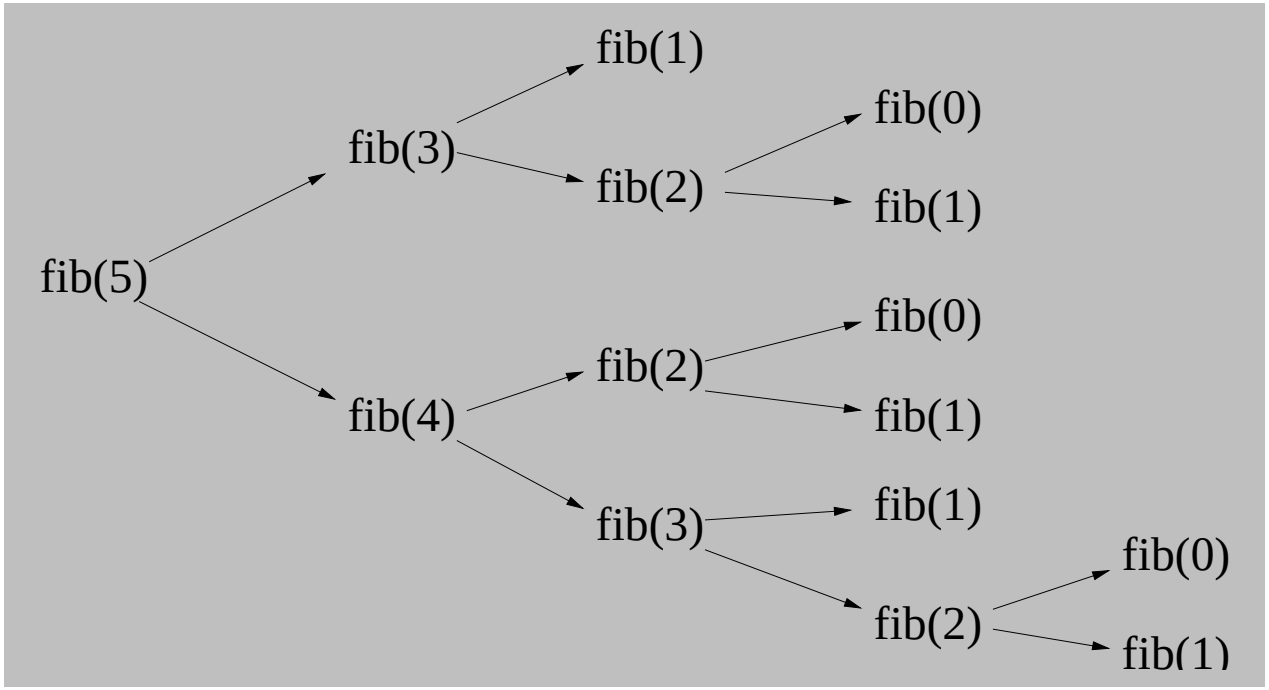
```
f0, f1  = 0, 1
for i in range(2,51):
    f = f0 + f1
    print(i, f)
    f0 = f1
    f1 = f
```

- **Result** (in less than a second)

```
2 1
3 2
...
49 7778742049
50 12586269025
```

Exponential Number of Calls

- Computing `fib(5)`



- fib(5) takes 15 calls to fib(N).
- fib(6) takes 25 calls to fib(N).
- fib(50) takes an enormous amount of calls to fib(N).
- All values between 1 and N $\Rightarrow$ the number is proportional to 2^N
  $\Rightarrow$ the computer crashes for $N = 50$.

Recursive functions (in general)

- A recursive method consists of:
  - One or more *base cases* where "simple" results are given explicitly.
  - One or more recursive rules (or steps) where "larger" results are expressed using "smaller" results.
- **Note**
  - We use *recursive rules* until a problem has been reduced to size where a base case can be used.
  - No base case ⇒ infinite recursion ⇒ program will crash.
- **Crash in practice**

```
# Start an infinite recursive call
def infinite(n):
    infinite(n+1)  # No base case ==> will never stop

# Program starts
infinite(0)
```

Result: The program runs for a second and crashes with a message `RecursionError: maximum recursion depth exceeded`

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)

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

Programming Example - Multiplication

Exercise

Write a recursive method `mult(a,b)` that computes the multiplication $a \cdot b$ with the use of addition. You can assume that both a and b are positive. Add also code that show how the recursive function `mult` can be used.

Solution idea

$$a \cdot b = \begin{cases} a & b = 1 \quad (\textit{base case}) \\ a + a \cdot (b-1) & b > 1 \quad (\textit{recursive step}) \end{cases}$$

- The recursive step decreases the value of b
- Hence, repeat the recursive step until we reach the base case

Solution - Recursive multiplication

```
def mult(a,b):
    if b == 1:
        return a
    else:
        return a + mult(a,b-1)

# Program starts
print( mult(3,7) )
print( mult(15,15) )
```

Recursive solutions are not that hard to understand. However, coming up with the solution idea takes a bit more practice.