Introduction to Python Programming

06 – Functions (Part I)

Josef van Genabith (Stefan Thater)

Dept. of Language Science & Technology

Universität des Saarlandes

WS 2022/23

Elements of imperative programs

- Expressions
  - Literals (numbers, strings, …)
  - Variables
  - Complex expressions
  - **Function calls**
- Statements
  - Assignments (var = expression)
  - **Function definitions** (actually, variable assignments)
  - Conditional statements (if ... elif ... else ...)
  - Loops (for, while)

Elements of imperative programs

- Why functions?

Factorial

```
x = 14
n = x
r = 1
while x > 0:
    r *= x
    x -= 1
print("The factorial of", n, "is", r)
```

Factorial

```python
x = 14
r = 1
for i in range(2, x + 1):
    r *= i
print("The factorial of", x, "is", r)
```

Functions

Functions are “subprograms” that can (and should) be used to divide a larger problem into several smaller problems.

```python
def factorial(x):
    '''Computes the factorial of x'''
    r = 1
    for i in range(2, x + 1):
        r *= i
    return r
```

Anatomy of a function definition

```
def name(var1, ..., varn):
    '''a short documentation, optional'''
    <code>
    return <something>
```

- **name**

  the name of the function (a variable)

- **var$_1$, ..., var$_n$**

  the parameters of the function

- **return <something>**

  usually at the end of the function definition, optional

Function definition

- Function definition = assignment to a variable

```
>>> def factorial(x):
...     '''Computes the factorial of x'''
...     r = 1
...     for i in range(2, x + 1):
...         r *= i
...     return r
...
>>> factorial
<function factorial at 0x1e57b0>
>>> help(factorial)
Computes the factorial of x
```

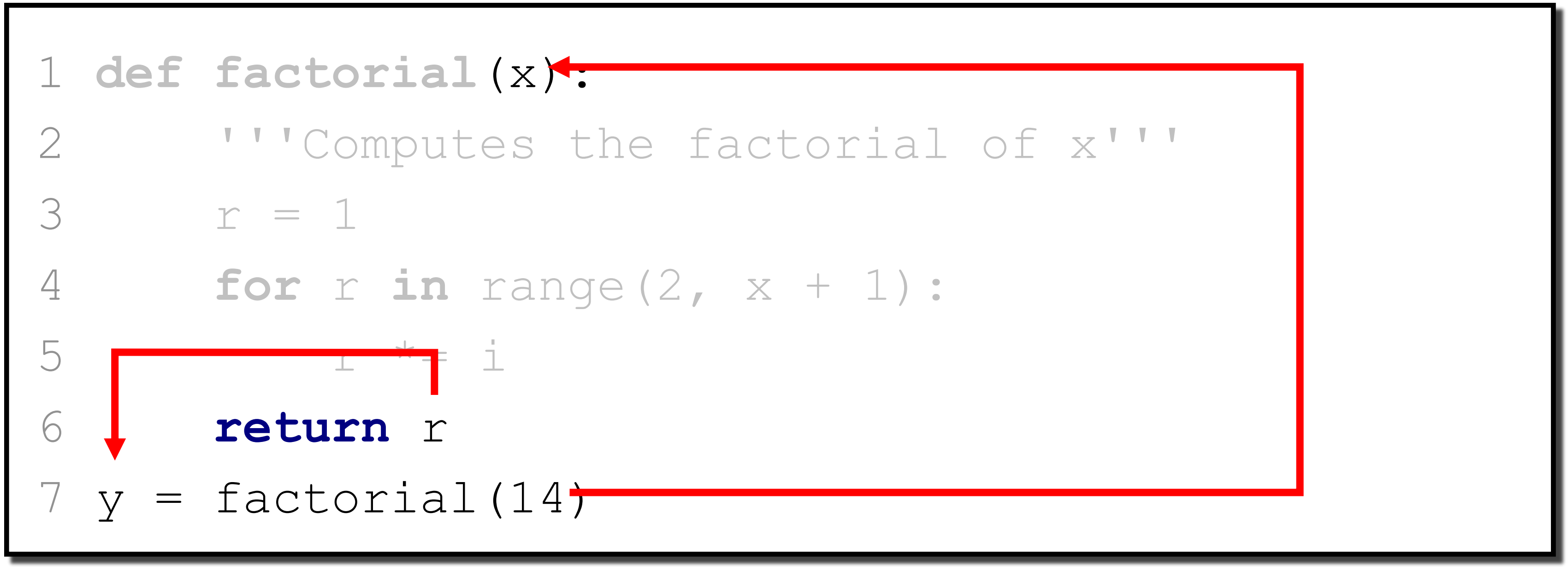
Function application

- If a function is applied, the code is executed

```python
def factorial(x):
    '''Computes the factorial of x'''
    r = 1
    for i in range(2, x + 1):
        r *= i
    return r

print(factorial(14)) # prints 87178291200
print(factorial(0))  # prints 1
```

Function application



```python
def factorial(x):
    '''Computes the factorial of x'''
    r = 1
    for r in range(2, x + 1):
        r *= i
    return r
y = factorial(14)
```

- When the function is called, the parameters are instantiated with the values from the function call
- The function call evaluates to the value returned by the function.

Exercise

- Implement a function **even()** that returns True when applied to an even number, False otherwise.

```
def even(...):
    <your code>
...
print(even(2)) # prints True
print(even(3)) # prints False
```

Answer

```python
def even(x):
    '''Returns True if x is even'''
    if x % 2 == 0:
        return True
    else:
        return False
```

```python
def even(x):
    '''Returns True if x is even'''
    return x % 2 == 0
```

The `return` statement

```
def binary(string):
    '''Returns True if all characters in string
       are 0 or 1'''
    for c in string:
        if c != '0' and c != '1':
            return False
    return True
```

- The `return` statement stops the execution of the function and returns a value
- The `return` statement can occur anywhere in the function definition (not just at the end)

The `return` statement

- Functions without return
  - equivalent to **`return None`** at the end of the function
- Naked return without a value
  - equivalent to **`return None`**
- Several values can be returned:
  - **return (value$_1$, ..., value$_n$)**
- Good programming style:
  - don't use naked return statements

return VS print

```
def binary1(string):
    for c in string:
        if c != '0' and c != '1':
            return False
    return True

def binary2(string):
    for c in string:
        if c != '0' and c != '1':
            print(False)
    print(True)
```

```
>>> s = '101101'
>>> binary1(s)
True
>>> binary2(s)
True
```

What's the difference?

return VS print

```
def binary1(string):
    for c in string:
        if c != '0' and c != '1
            return False
    return True

def binary2(string):
    for c in string:
        if c != '0' and c != '1':
            print(False)
    print(True)
```

```
>>> s = '101101'
>>> x = binary1(s)
>>> x
True
>>> y = binary2(s)
True
>>> y
>>> # None, not printed
```

Exercises #1 and #2

- Turn your code of from the last exercises into functions:
- Implement a function `is_prime(x)` that returns `True` if `x` is prime, `False` otherwise
  - `is_prime(7)` ⇒ `True`
  - `is_prime(15)` ⇒ `False`
- Implement a function `gcd(x, y)` that computes the greatest common divisor of `x` and `y`.
  - `gcd(8, 12)` ⇒ `4`

Exercise #3

- Implement a function `is_member(x, list)` that returns `True` if `x` is an element of `list`, `False` otherwise
  - `is_member(2, [1, 2, 3])` ⇒ `True`
  - `is_member(4, [1, 2, 3])` ⇒ `False`
- Note that this is exactly what the `in` operator (provided by Python) does.
- For the sake of the exercise you should not use the `in` operator …

Exercise #4

- Implement a function that computes the intersection of two lists, i.e. a function that returns a list of elements that are members of both input-lists.
  - `intersection([1, 2, 3, 4], [2, 4, 6])` ⇒ `[2, 4]`
- Hints and comments:
  - `x = []` creates an empty list
  - `x.append(y)` adds `y` to list `x`

Exercise #5

- Implement a function that recognizes palindromes
  - `is_palindrome("level")` ⇒ `True`
  - `is_palindrome("levels")` ⇒ `False`
- Hints:
  - `string[i]` ⇒ ith character from left (starting from 0)
  - `string[-i]` ⇒ ith character from right (starting from 1)
  - Integer division: `5 // 2` ⇒ `2`