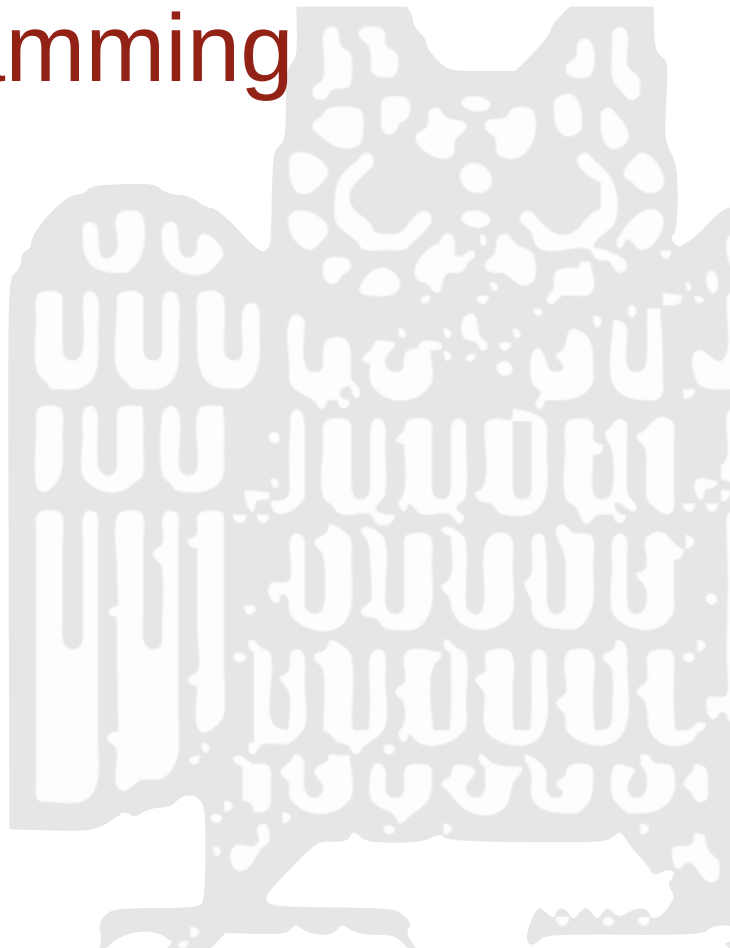
Introduction to Python Programming

10 – Functions (Part II)

Josef van Genabith (Stefan Thater)
Dept. of Language Science & Technology
Universität des Saarlandes

WS 2022/23

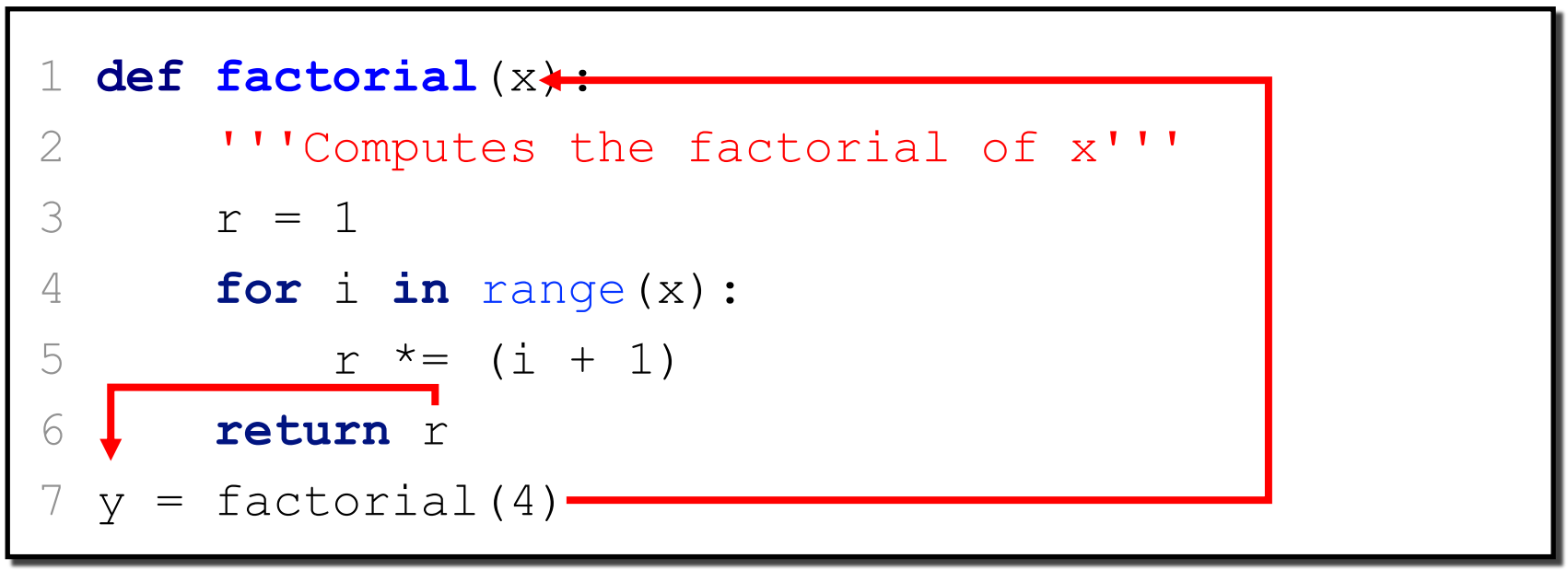


Functions – Recap

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

```
1 def factorial(x):
2     '''Computes the factorial of x'''
3     r = 1
4     for i in range(x):
5         r *= (i + 1)
6     return r
7 y = factorial(4)
```

Function application



- 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.

Local variables

```
1 def factorial(x):
2     '''Computes the factorial of x'''
3     r = 1
4     for i in range(x):
5         r *= (i + 1)
6     return r
7 y = factorial(4)
```

- Functions introduce **local variables**
  - Parameters
  - Variables to which a value is assigned
- Local variables are not visible outside of the function !

Local variables

```
1 def factorial(x):
2     '''Computes the factorial of x'''
3     r = 1
4     for i in range(x):
5         r *= (i + 1)
6     return r
7 y = factorial(4)
8 print(x)  # ???  Similar for print(r), print(i)…
```

- Functions introduce **local variables**
  - Parameters
  - Variables to which a value is assigned
- Local variables are not visible outside of the function !

Variables and Namespaces

- Variables live in namespaces
  - Namespaces map variables to their values
- Namespaces can be nested
  - Function calls create local namespaces, which are embedded within the namespace of the calling context
- **Variables with the same name in different namespaces can refer to different values**
  - Local variables "shadow" ("hide"!) non-local (global) variables
  - If I declare them locally! That is, if I use them in an explicit assignment statement, or if they are parameters of the function!

Variables and Namespaces

```
1  r = 'something'
2  def factorial(x):
3      '''Computes the factorial of x'''
4      r = 1
5      for i in range(x):
6          r *= (i + 1)
7      return r
8  y = factorial(4)
9  print(y) # 24
10 print(r) # something
```

- Local variables “shadow” non-local (global) variables

Variables and Namespaces

```
1  r = 'something'
2  def factorial(x):
3      '''Computes the factorial of x'''
4      r = 1
5      for i in range(x):
6          r *= (i + 1)
7      return r
8  y = factorial(4)
9  print(y) # 24
10 print(r) # something
```

- Local variables “shadow” non-local (global) variables

Variables and Namespaces

```
1  r = 'something'
2  def factorial(x):
3      '''Computes the factorial of x'''
4      r = 1
5      for i in range(x):
6          r = r *(i + 1)
7      return r
8  y = factorial(4)
9  print(y) # 24
10 print(r) # something
```

- Local variables “shadow” non-local (global) variables

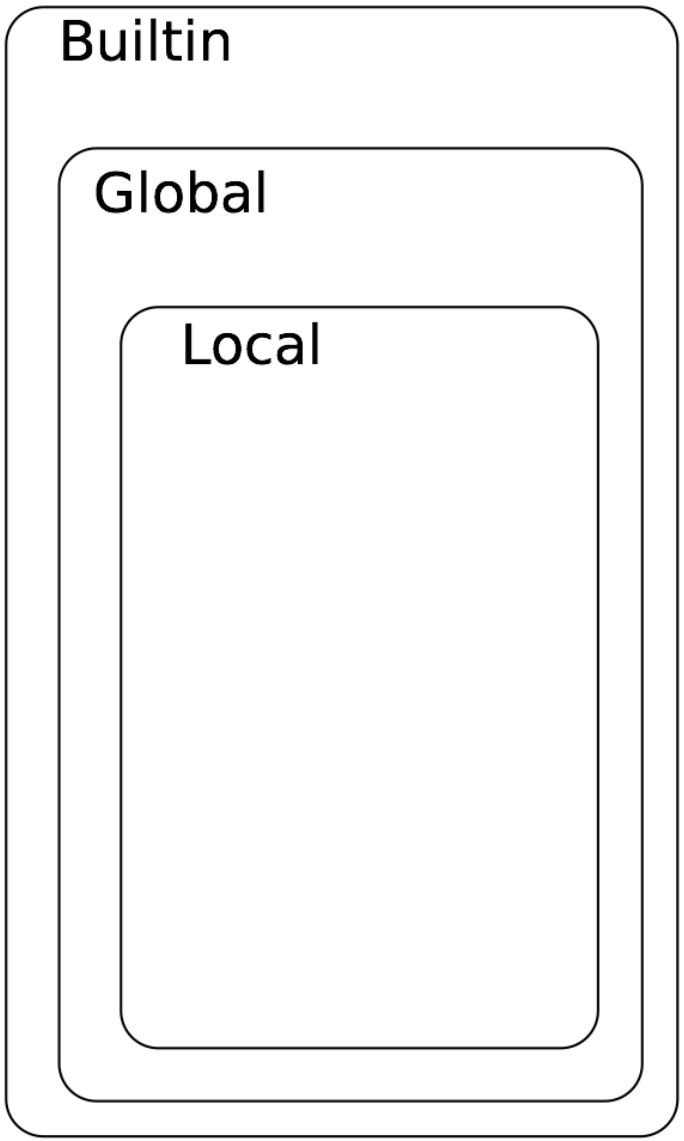
Variables and Namespaces

```
1  r = 'something'
2  def factorial(x):
3      '''Computes the factorial of x'''
4      #r = 1
5      for i in range(x):
6          r = r *(i + 1)
7      return r
8  y = factorial(4)
9  print(y) # 24
10 print(r) # something
```

- Local variables “shadow” non-local (global) variables
- What is going to happen now with `#r = 1` ???

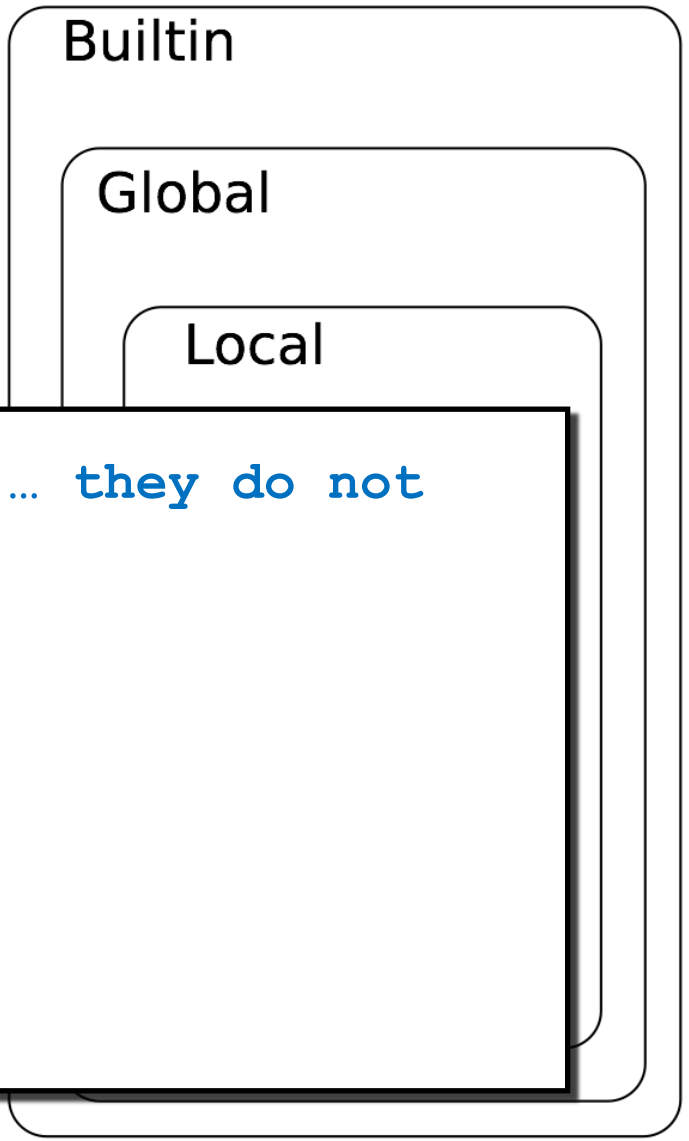
Variables and Namespaces (functions!)

- Built-in namespace
  - created when Python starts
  - contains built in names (e.g., abs)
- Global namespace
  - created when the program is executed (read in)
  - contains the “top-level” names
- Local namespace
  - created when a function (!) is called
  - contains the local variables



Variables and Namespaces (functions!)

- Builtin namespace
  - created when Python starts
  - contains built in names (e.g., abs)
- Global namespace

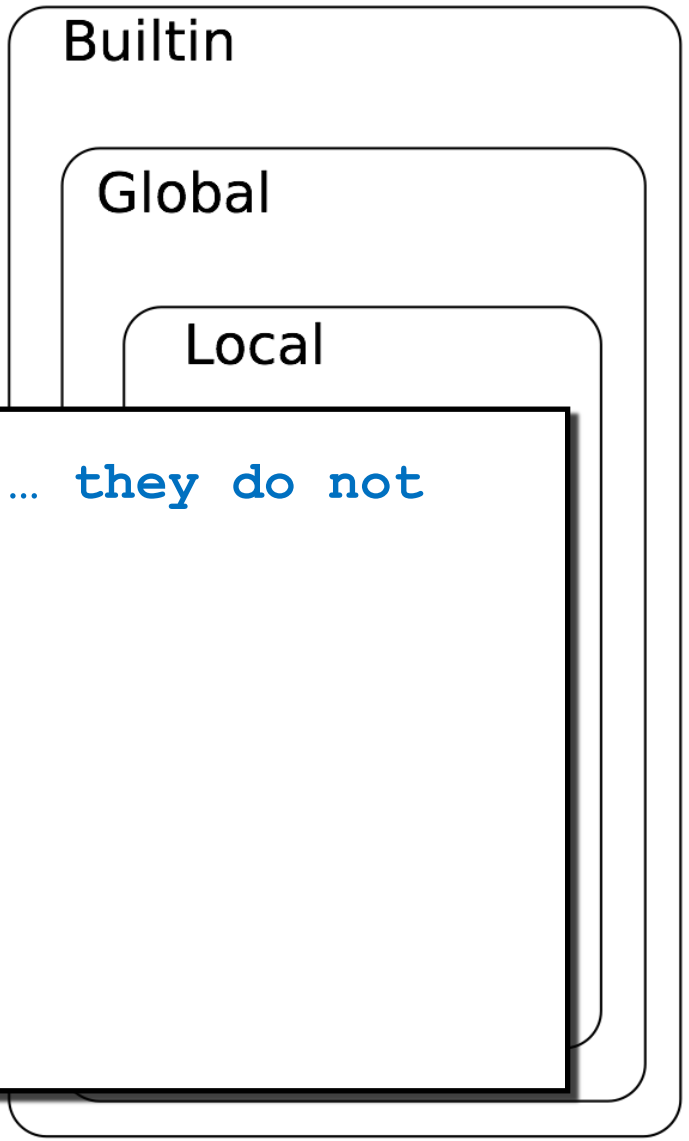


Not for loops, branching commands … they do not create namespaces

```
1   r = 1
2   myList = [1,2,3]
3   for el in myList:
4       r = r + el
5   print(r)
```

Variables and Namespaces (functions!)

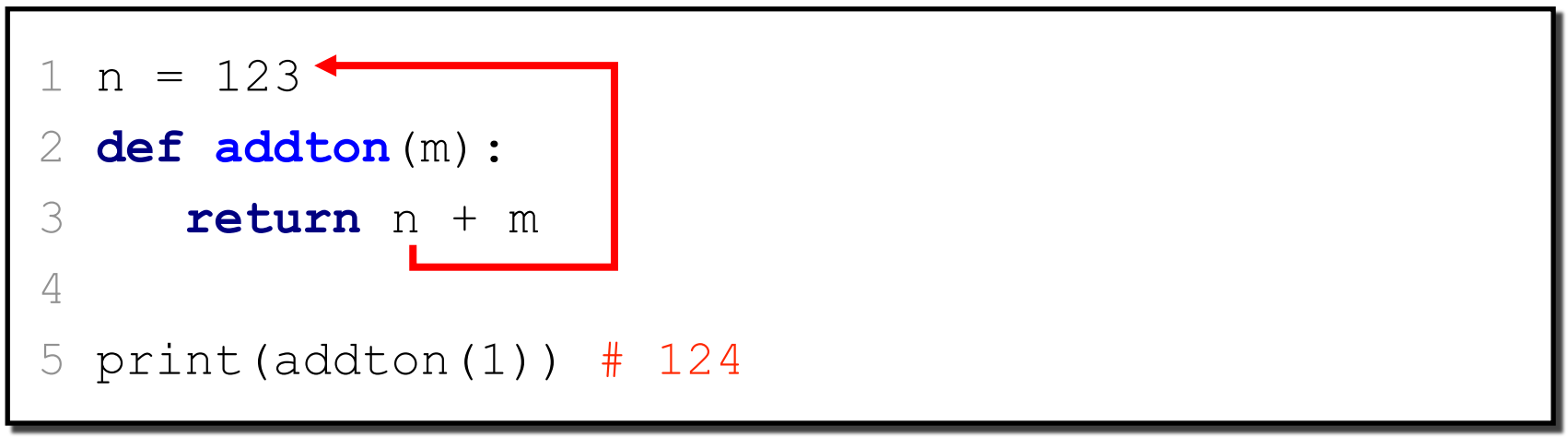
- Builtin namespace
  - created when Python starts
  - contains built in names (e.g., abs)
- Global namespace



Not for loops, branching commands … they do not create namespaces

```
1  r = 1
2  myList = [1,2,3]
3  for el in myList:
4      r = r + el
5  print(r)
```

Local and global variables



- Within a function, we can access (i.e. read the value of) global (non-local) variables …

Local and global variables

```
1 n = 123
2 def addton(m):
3     n = n + m
4
5 print(addton(1))
UnboundLocalError: local variable 'n' referenced
before assignment
```

n is a local variable because we assign a value to it

- Within a function, we can access (read the value of) global (non-local) variables …
- But **we cannot assign values** to a non-local variable
  - (Modifying the value of a non-local variable is possible)

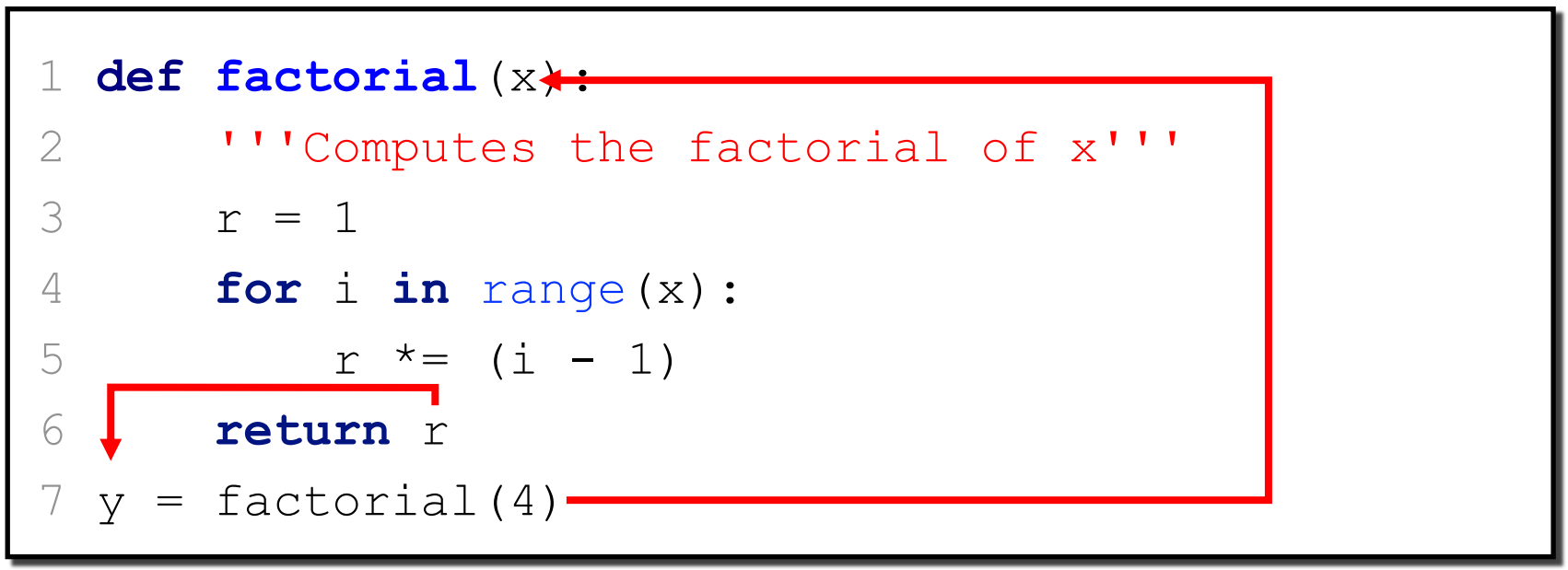
Local and global variables

```
1 n = 123
2 def addton(m):
3     n = n + m
4
5 print(addton(1))
UnboundLocalError: local variable 'n' referenced
before assignment
```

n is a local variable because we assign a value to it

- Within a function, we can access (read the value of) global (non-local) variables …
- But **we cannot assign values** to a non-local variable
  - (Modifying the value of a non-local variable is possible)

Function application



- 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.

Parameter Passing

- **Call by value** (some other programming languages)
  - pass a copy of the value to the function
- **Call by reference** (Python, Java, …)
  - pass a reference (“pointer”) to the value to the function
  - the function (!) can modify the value!

Side effects – another example

side effect = changing the value of a global variable (roughly …)

```
1 def incr(items):
2     for i in range(len(items)):
3         items[i] = items[i] + 1
4     return items
5
6 example = [1,2,3,4]
7 print(example)          # [1,2,3,4]
8 print(incr(example))    # [2,3,4,5]
9 print(example)          # [2,3,4,5]
```

No side effect

Here we assign a new value to n
(we do not modify the value)

```
1 def incr(n):
2     n += 1
3
4 example = 1
5 print(example) # prints 1
6 incr(example)
7 print(example) # prints 1
```

No side effect

Here we assign a new value to n
(we do not modify the value)

```
1 def incr(n):
2     n += 1
3
4 example = 1
5 print(example) # prints 1
6 print(incr(example)) # prints ____?
7 print(example) # prints 1
```

No side effect

Here we assign a new value to n
(we do not modify the value)

```
1 def incr(n):
2     n += 1
3
4 example = 1
5 print(example) # prints 1
6 print(incr(example)) # prints None
7 print(example) # prints 1
```

No side effect

Here we assign a new value to n
(we do not modify the value)

```
1 def incr(n):
2     n += 1
3     return n
4 example = 1
5 print(example) # prints 1
6 print(incr(example)) # prints _____?
7 print(example) # prints 1
```

No side effect

Here we assign a new value to n
(we do not modify the value)

```
1 def incr(n):
2     n += 1
3     return n
4 example = 1
5 print(example) # prints 1
6 print(incr(example)) # prints 2
7 print(example) # prints 1
```

Style guide

- Try to avoid side effects (when possible)!
- Functions with side effects should not return a value

Recursion

- Functions can call other functions
- Functions can also call themselves
  - This is called **recursion**
- Many problems can be elegantly solved using recursion

An Example

- The Fibonacci sequence is an infinite sequence of numbers where each number is found by adding up the two numbers before it:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . .
  - 0, 1
  - 0, 1, 1
  - 0, 1, 1, 2
  - 0, 1, 1, 2, 3
  - 0, 1, 1, 2, 3, 5
  - …

An Example

- The Fibonacci sequence is an infinite sequence of numbers where each number is found by adding up the two numbers before it:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . .
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, . . .

An Example

- The Fibonacci sequence is an infinite sequence of numbers where each number is found by adding up the two numbers before it:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . .
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, . . . (i-th Fibonacci Number)

```
(BC) fib(0)= 0     fib(1)= 1
(RC) fib(n) = fib(n-1)+fib(n-2)   for n > 1
```

An Example

- The Fibonacci sequence is an infinite sequence of numbers where each number is found by adding up the two numbers before it:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . .
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, . . .

```
1 def fib(n):
2     if n <= 1:
3         return n
4     else:
5         return fib(n-1) + fib(n-2)
```

Exercises

- Write a recursive function computing the factorial of a non-negative number
  - fak(0) = 1
  - fak(n) = n * fak(n - 1)
- Implement a version of sum() that computes the sum of numbers in a possibly nested list of lists

```
>>> nestedsum([1, 2, 3, 4, 5])
15
>>> nestedsum([1, [2, 3, [4], []], [5]])
15
```

Answers

Answers

```
1 def fak(n):
2     if n == 0:
3         return 1
4     else:
5         return n * fak(n - 1)
```

Functions inside functions

```
1 def outer(x):
2     def inner(y):
3         return x + y
4     return inner(1)
```

- Functions can be defined inside functions
- Often used to implement helper functions that performs some of the computations of another function

Functions inside functions

```
1 def outer(x):
2     def inner(y):
3         return x + y
4     return inner
5 f1 = outer(1)
6 f2 = outer(2)
7 print(f1(3)) # prints 4
8 print(f2(3)) # prints 5
```

- Functions can return also return other functions

Keyword arguments

default value

```
1 def sqrt(x, precision = .00001 ):
2     '''Computes the square root of x'''
3     g = x
4     while (g * g) - x > precision:
5         g = (g + x / g) / 2
6     return g
```

```
>>> sqrt(2)
1.4142156862745097
>>> sqrt(2, precision = .01)
1.4166666666666665
>>> sqrt(2, .01)
1.4166666666666665
```

Achtung!

- The default value is evaluated when the function definition is evaluated (read in)
- This can have very strange effects when the default value is a list (or some other value that can be modified)

Achtung!

default value

```
1 def achtung(someparameter = []):
2     someparameter.append(1)
3     return someparameter
4
5 print(achtung())        # [1]
6 print(achtung())        # [1, 1]
7 print(achtung([]))      # [1]
```

What happens here?

```
1 def achtung(someparameter = []):
2     someparameter.append(1)
3     return someparameter
4
5 x = [2]
6 print(achtung(x)) # prints what?
7 print(achtung(x)) # prints what?
```

More about functions

- Many more (not so important) details about functions
  - see www.python.org

More exercises

- Implement a non-recursive version of fib. Compare runtimes for inputs 10, 20, 30, 35

```
def fibit(number):
    <your code>

print(fibit(10))
# 55
```

More exercises

- Implement a recursive function perm that computes all permutations of an input string:

```
def perm(strng):
    <your code>

print(perm("abc"))
# ['abc', 'bac', 'bca', 'acb', 'cab', 'cba']
```

More exercises

- Modify the naive implementation of a sorting algorithm so that the list to be sorted is modified **in place**.
- Here is the idea:
  1. Iterate over all indices i = 0, 1, 2, ... of the input list
  2. Find the index j of the smallest element in the rest of the list (from i on)
  3. Replace the elements at indices i and j