EINFÜHRUNG IN DIE PROGRAMMIERUNG

FUNCTIONS

DHBW MANNHEIM
WIRTSCHAFTSINFORMATIK (DATA SCIENCE)

Markus Menth
Martin Gropp

AGENDA

DEFINING FUNCTIONS

- Python Tutorial - Defining Functions

AGENDA

ARGUMENTS

- Default Argument Values
- Keyword Arguments
- Arbitrary Argument Lists
- Unpacking Argument Lists

AGENDA

FUNCTIONAL PROGRAMMING

- Lambda Expressions

DOCUMENTATION AND ANNOTATION

- Documentation Strings

FUNCTIONS

MOTIVATION

SUBROUTINE

*In computer programming, a **subroutine is a sequence of program instructions that perform a specific task, packaged as a unit**. This unit can then be used in programs wherever that particular task should be performed. Subprograms may be defined within programs, or separately in libraries that can be used by multiple programs. In different programming languages, a subroutine may be called a procedure, a function, a routine, a method, or a subprogram. (Source: **Wikipedia**)*

FUNCTION

Generally speaking, ***a function is a "subprogram" that can be called by code external (or internal in the case of recursion) to the function****. Like the program itself,* ***a function is composed of a sequence of statements called the function body****.* ***Values can be passed*** *to a function, and the* ***function can return a value.*** *(Source* ***MDN****)*

MOTIVATION

ADVANTAGES OF FUNCTIONS

- A program is decomposed into smaller and simpler steps
- Reduces code duplication and allows code-reuse across different projects
- Allows to assign small pieces of work to team members
- Provides a contract between user of a function and implementer (the signature of the function)
- Hides implementation details from the user of a function
- Improves readability and maintainability

MOTIVATION

DISADVANTAGE: OVERHEAD

- Function calls take more time
- Typically outweighed by the above-mentioned advantages

DEFINING FUNCTIONS

Functions are defined using the **def** keyword:

```
def ⟨name⟩(⟨parameters⟩):
        ⟨body⟩
```

- *⟨name⟩*: a unique identifier,
- *⟨parameters⟩*: parameters separated by commas: variable names that receive values when the function is called
- *⟨body⟩*: the statements that are executed when the function is called

DEFINING FUNCTIONS

EXAMPLE

```
def my_function(a, b):
        print(f'My arguments were {a} and {b}')
```

Additional documentation: **Python Tutorial - Defining Functions**

CALLING/INVOKING FUNCTIONS

When calling a function, you need to supply values for their required parameters.

- Parameter values are assigned by position (positional parameters)
- A value for each parameter is required

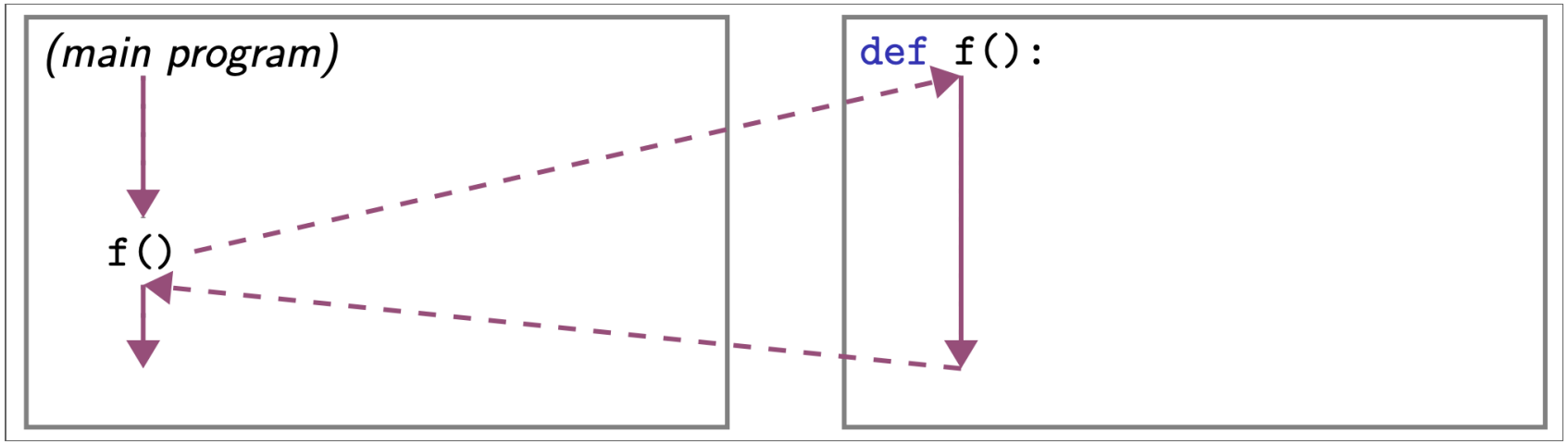
Parameters are like local variables inside the function.

```
def my_function(a, b):
        print(f'My arguments were {a} and {b}')

my_function(1, 2)
```

```
My arguments were 1 and 2
```

CONTROL FLOW



CALLING/INVOKING FUNCTIONS

When you call a function, you can also use more complex expressions for the arguments:

```
x = 10
my_function(x, x + 1)
```

```
My arguments were 10 and 11
```

⚠ Parameter names and variable names used for calling the function have nothing to do with each other!

⚠ Assigning new values to parameters inside the function has no effect on variables used to call the function.

(Details on that later.)

EXAMPLE

```
def foo(n):
  print(n * 2)
```

```
foo(10)
i = 20
foo(i)
# Does not work: fib()
```

RETURN VALUES

Functions can **return** a value:

```
def add_one(x):
        return x + 1
```

```
# assign to a variable
a = add_one(10)

# use in an expression
b = 2 * add_one(10)

# even as an argument to another function
print(add_one(10))

# or simply ignore it
add_one(10)
```

Returning a value from a function is not the same as `printing` it!

RETURN VALUES

A *return* statement ends the function.

```
def f(x):
        if x < 0:
                return

        print(x)

f(-1)
```

(no output)

RETURN VALUES

Python functions always return a value

- The value is passed back via the return keyword
- `return` expression exits a function immediately (returns to the caller)
- Using `return` without a value is equivalent to `return None`
- If no return expression exists, the function returns `None`

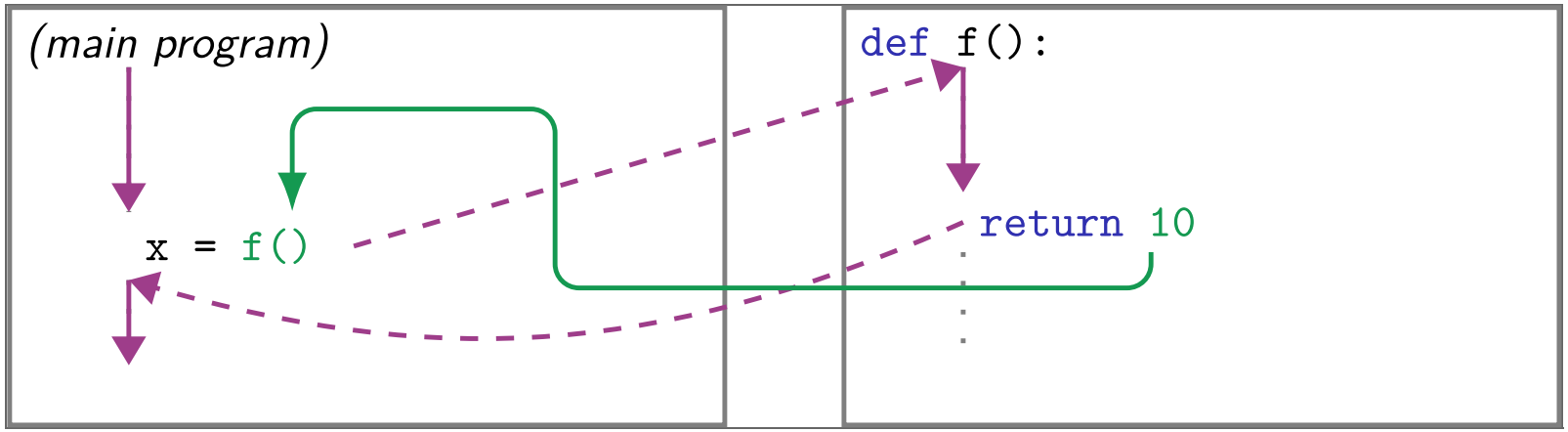
RETURN VALUES

EXAMPLE

```
def bla(x):
        if x is None:
                return
        elif x == 1:
                return 10
        return 11

print(bla(None))
print(bla(1))
print(bla(2))
```

CONTROL FLOW



TYPE HINTS

TYPE HINTS

```
def f(s, x):
        print(f'Mein Name ist {s}')
        print(f'Der Wert multipliziert mit 3 ist: {3*x}')

f(17, 'Martin')
```

```
Mein Name ist 17
Der Wert multipliziert mit 3 ist: MartinMartinMartin
```

TYPE HINTS

To make it easier to spot mistakes in function calls, Python supports **type hints** for parameters and return values.

```
def f(s: str, x: float) -> bool:
        return True
```

Python itself does not enforce type constraints, but a good Python IDE will highlight these type conflicts.

```
f(42, 'foo')  # your IDE should warn you here!
```

TYPE HINTS

Some important types:

- `int`
- `float`
- `str`
- `bool`

Functions returning `None` are annotated with the return type `None`.

TYPE HINTS

For container types you should import the matching types from Python's **typing** module.

- `List`
- `Set`
- `Tuple`
- `Dict`

TYPE HINTS

The types inside the containers are specified in square brackets:

- `List[float]`
- `Tuple[List[int], int, str]`
- `Dict[str, str]` etc.

The types have to be imported first:

```
from typing import List

def print_sum(x: List[int]) -> None:
        print(sum(x))
```

(Python 3.9+ supports `list[T]` etc., importing from typing is no longer necessary there.)

TYPE HINTS

EXAMPLE

```
from typing import Dict, Set

def foo(d: Dict[str, int]) -> Set[int]:
        return set(d.values())

def main() -> None:
        foo({'a': 1, 'b': 2, 'c': 1})
```

TYPE HINTS

For tuples, the type of each element is specified:

```
from typing import Tuple

def bar(t: Tuple[int, str]) -> Tuple[str, int]:
        return t[1], t[0]

t1 = (17, 'foo')
t2 = bar(t1)
```

SPECIAL ARGUMENTS

DEFAULT ARGUMENTS

Convenience: for each argument, a default value can be provided.

```
def verify_password(
        password: str,
        prompt: str = 'pw> ',
        retries: int = 3,
        reminder: str = 'Wrong password, try again...'
) -> bool:
        for t in range(retries):
                in_pw = input(prompt).strip()
                if in_pw == password:
                        return True
                print(reminder)

        return False
```

```
verify_password('geheim!!!elf!')
```

```
verify_password('geheim!!!elf!', 'PW: ', 2)
verify_password('geheim!!!elf!', 'PW: ', 2, 'Falsch')
```

Additional documentation: **Default Argument Values**

DEFAULT ARGUMENTS: CAVEAT

Important: Default values are evaluated only once!

⚠ What happens when using *mutable* objects (lists, dicts, etc.)?

EXAMPLE ☠

```
def bla(a, l=[]):
    l.append(a)
    return l

print(bla(1))
print(bla(2))
print(bla(3))
```

Any ideas for a workaround?

DEFAULT ARGUMENTS: CAVEAT

WORKAROUND

```
def bla(a, l=None):
    if l is None:
        l = []

    l.append(a)
    return l
```

DEFAULT ARGUMENTS: CAVEAT

What happens here?

```
def bla(a, l=[]):
    l.clear()
    l.append(a)
    return l

a = bla(1)
b = bla(2)
c = bla(3)

print(a)
print(b)
print(c)
```

DEFAULT ARGUMENTS: CAVEAT

What happens here?

```
def bla(a, l=[]):
    l.clear()
    l.append(a)
    return l

a = bla(1)
b = bla(2)
c = bla(3)

print(a)
print(b)
print(c)
```

```
[3]
[3]
```

```
[3]
```

KEYWORD ARGUMENTS

An alternative to positional arguments.
Keyword arguments follow positional arguments.

EXAMPLE

```
foo(1, bar=2, baz="3")
```

Additional documentation: **Keyword Arguments**

KEYWORD ARGUMENTS

EXAMPLE

```
def verify_password(
    password: str,
    prompt: str = "pw> ",
    retries: int = 3,
    reminder: str = "Wrong password, try again..."
) -> bool:
    for t in range(retries):
        in_pw = input(prompt)
        if in_pw.strip() == password:
            return True
        print(reminder)

    return False

verify_password("lala")
verify_password("lala", retries = 5)
```

```
# Wrong: verify_password("lala", "PW:" , prompt = "Lala: ")
# Wrong: verify_password(retries = 4, "lala")
```

KEYWORD ARGUMENTS

Instead of listing individual parameters, a dict can be used

- Must be the final parameter in the form `**kwargs`
- This dict contains all keyword arguments (except for those corresponding to a formal parameter)

EXAMPLE

```
def bla(msg, **kwargs):
    print(msg)
    for kw in kwargs:
        print(kw, ":", kwargs[kw])

bla("Hallo", a=1, b=5, blubb="Hallo", eine_liste=[])
```

ARBITRARY ARGUMENT LISTS

Allows defining functions that can be called with an arbitrary number of arguments

- These are NOT keyword arguments but formal positional parameters
- Defined using `*args`, parameters are passed as a tuple
- Before the variable number of arguments, zero or more normal arguments may occur

```
def foo(*args):
        print('Number of arguments:', len(args))
```

ARBITRARY ARGUMENT LISTS

EXAMPLE

```
def bla(*args, lala=1234):
        print("lala", lala)
        print("args", args)

bla(1,2,3,4)
bla(1,2,3,4, lala="Hallo")
```

Additional documentation: **Arbitrary Argument Lists**

UNPACKING ARGUMENT LISTS

Sometimes arguments are already available as a list or dict.
* unpacks positional arguments from a list or tuple.

EXAMPLE

```
def foo(a, b):
    print(a, b)

params = [1, 2]
foo(*params)
```

UNPACKING ARGUMENT LISTS

** unpacks keyword arguments from a dict.

EXAMPLE

```
def foo(**kwargs):
    print(kwargs)

params = {"a": 1, "b": 2}
foo(**params)
```

Additional documentation: **Unpacking Argument Lists**

GLOBAL VARIABLES

GLOBAL VARIABLES

Using global variables introduces side effects: a function's behavior depends on global state.
Before you can write to global variables, you need to confirm this with the `global` keyword.

EXAMPLE

```
i = 10
def foo():
    global i
    i += 1
    print(i)

foo()
foo()
```

Try to avoid using global variables!

FUNCTIONAL PROGRAMMING

PURE FUNCTIONS

Functions with side effects are sometimes called **procedures**.
Pure functions, on the other hand, behave like mathematical functions, i.e., their return value only depends on their parameters and they have no side effects.

FUNCTIONAL PROGRAMMING

Functional programming is a ***programming paradigm*** *— a style of building the structure and elements of computer programs — that treats computation as the evaluation of mathematical functions and* ***avoids changing-state and mutable data****.*
It is a declarative programming paradigm, which means programming is done with expressions or declarations instead of statements. In functional code, the output value of a function depends only on the arguments that are passed to the function, so ***calling a function f twice with the same value for an argument x produces the same result f(x) each time***
Eliminating side effects*, i.e., changes in state that do not depend on the function inputs, can make it much* ***easier to understand and predict the behavior of a program****, which is one of the key motivations for the development of functional programming.*
Source: ***Wikipedia - Functional Programming***

FUNCTIONS ARE FIRST-CLASS CITIZENS

Functions are first-class citizens of the language.

- They can be used just like other elements (e.g., numbers, strings, ...)
- They can be used like any other variable definition
- They can even be assigned to other variables.

FUNCTIONS ARE FIRST-CLASS CITIZENS

EXAMPLE

```
def foo():
    print("Bla")

x = foo

foo()
x()

print(foo)
print(x)
```

FUNCTIONS ARE FIRST-CLASS CITIZENS

Functions are first-class citzens of the language

- They can be used just like other elements (e.g., numbers, strings, ...)

CONSEQUENCES

1. Functions can be defined in functions
2. Functions can be passed to functions as parameters
3. Functions can be returned from functions

CONSEQUENCE #1: FUNCTIONS IN FUNCTIONS

Functions can be defined in functions

- Inner function captures the environment of the outer function
- So-called closure

```
def outer(num1):
    def inner_increment(num1):  # Hidden from outer code
        return num1 + 1
    num2 = inner_increment(num1)
    print(num1, num2)

# Does not work: inner_increment(10)
outer(10)
```

CONSEQUENCE #2: FUNCTIONS AS PARAMETERS

Functions can be passed to functions as parameters

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]

pairs.sort()
print(pairs)
```

```
def sort_criterion(pair):
        return pair[1]

pairs.sort(key=sort_criterion)
print(pairs)
```

CONSEQUENCE #3: FUNCTIONS AS RETURN VALUES

Functions can be returned from functions

```
def make_multiplier(n):
        def f(x):
                return x * n
        return f

mult_3 = make_multiplier(3)
mult_5 = make_multiplier(5)

print(mult_3(10))
print(mult_5(10))
```

LAMBDA EXPRESSIONS

Syntactic sugar to create small, anonymous functions

- Small: limited to a single expression
- Anonymous: lambda functions have no name

Can be used whenever a function object is required

- Created via the **lambda** keyword
- E.g. as a parameter or return value

Additional documentation: **Lambda Expressions**

LAMBDA EXPRESSIONS: EXAMPLES

EXAMPLE 1

Pass a lambda expression as a parameter

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[1])
print(pairs)
```

LAMBDA EXPRESSIONS: EXAMPLES

EXAMPLE 2

Return a lambda expression from a function

```
def make_multiplier(n):
        return lambda x: x * n

mult_3 = make_multiplier(3)
mult_5 = make_multiplier(5)

print(mult_3(10))
print(mult_5(10))
```

LAMBDA EXPRESSIONS: EXAMPLES

```
a = list(range(10))
b = []
for x in a:
        if x % 2 == 0:
                b.append(x**2)
```

```
a = list(range(10))
b = list(map(
        lambda x: x**2,
        filter(
                lambda x: x % 2 == 0,
                a
        )
))
```

LAMBDA EXPRESSIONS: EXERCISE

Convert this program to use a list comprehension to compute b!

```
a = list(range(10))
b = list(map(
        lambda x: x**2,
        filter(
                lambda x: x % 2 == 0,
                a
        )
))
```

LAMBDA EXPRESSIONS: EXERCISE

```
a = list(range(10))
b = list(map(
        lambda x: x**2,
        filter(
                lambda x: x % 2 == 0,
                a
        )
))
```

```
a = list(range(10))
b = [
x**2
        for x in a
        if a % 2 == 0
]
```

DOCUMENTATION STRINGS

DOCUMENTATION STRINGS

Python functions can be documented by putting a string at the beginning of a function's body.
It is called a *docstring* and can be displayed with the `help` command or converted to HTML or PDF documents using external tools like *Sphinx*.

```
def foo():
        """
        This function does nothing.

        But it's really good at it!
        """
        pass
```

DOCUMENTATION STRINGS

CONVENTIONS

- First line contains a concise summary
- Second line should be blank (separating summary from the rest)
- Following lines uses one or more paragraphs to describe the function
- Additional documentation **here** and **here**

PARAMETER PASSING

DETAILS OF PARAMETER PASSING: CALL BY VALUE

In Python, parameters are passed "by value":

- the value is **copied** to the parameter of the invoked function
- assignments to the parameter inside the function are *not* visible outside the function

(Older versions of these slides contained wrong information about this topic.)

EXAMPLE: CALL BY VALUE

```
def myfunc(i):
        i = 5
        print("Inside the function: i = ", i)

i = 10
print("Before invoking myfunc: i = ", i)
myfunc(i)
print("After invoking myfunc: i = ", i)
```

```
Before invoking myfunc: i = 10
Inside the function: i = 5
After invoking myfunc: i = 10
```

MUTABLE OBJECTS

When a mutable object (e.g. a list) is passed as a parameter, the object itself can be modified!

```
def myfunc(i):
        i.append("bla")
        print("Inside the function: i = ", i)

i = [10]
print("Before invoking myfunc: i = ", i)
myfunc(i)
print("After invoking myfunc: i = ", i)
```

```
Before invoking myfunc: i = [10]
Inside the function: i = [10, "bla"]
After invoking myfunc: i = [10, "bla"]
```

❗ This is different from *assigning* (=) a new value to `i`!

BUT:

```
def myfunc2(i):
        i = ["bla"]
        print("Inside the function: i = ", i)

i = [10]
print("Before invoking myfunc2: i = ", i)
myfunc2(i)
print("After invoking myfunc2: i = ", i)
```

```
Before invoking myfunc: i = [10]
Inside the function: i = ["bla"]
After invoking myfunc: i = [10]
```

Assignments to a parameter (like `i = ["bla"]`) are never visible for the caller.

In other programming languages, this can be achieved using Call by Reference.

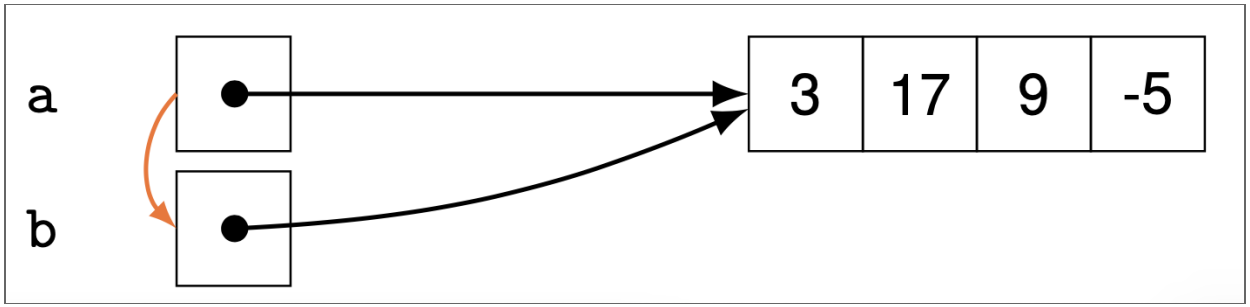
WHY?

👉**References**

Python passes a *reference* to the object *by value*.

The object itself is not copied.

The copy of the reference will still refer to the same object.



EXERCISES

Implement the **Caesar cipher** as a function

- The function accepts a text and a (positive or negative) number representing the shifts

EXERCISE

Implement a function that computes the **Body Mass Index**

- Accept all variables as parameters
- Write a second functioon that returns the category of a BMI value (e.g., "Severely underweight")

EXERCISE

Convert the following excercises from the last chapter to use functions

- Reverse the order of a string
- Caesar cipher (implement `encrypt`, `decrypt`, and `shift_chars`)
- Greatest Common Divisor
- Fibonacci numbers

EXERCISES

Write a function which calculates and then displays the average, minimum, median, and maximum values of a series of temperature readings stored in an array of numbers

- Implement a function for each of the computations and one to display the accumulated information

EXERCISES

Write a function that displays christmas trees using * characters only

- The tree's height (in lines) is passed as a parameter; use a default of 7 if no parameter is passed
- The minimum height is 4 (including the tree trunk); display a warning on stderr and return False in all other cases
- Invoke the function once without passing a parameter, once with a value of 5, and once with −1

The output looks like this for height 7:

```
      *
     ***
    *****
   *******
  *********
      *
      *
```

EXERCISE

Implement a function with the following properties

- It uses **bubble sort** to sort a list, which is passed as a parameter
- Accepts an optional keyword parameter that provides a sort criterion as a function. It returns 0 for equal parameters, -1 for `a < b` and 1 for `a > b`

Implement a second function that calls the first function but accepts an arbitrary argument lists and the sort criterion using a keyword parameter

EXERCISE

Implement the **Sieve of Eratosthenes** as a function

SOLUTION

```
a = list(range(101))
a[0] = None
a[1] = None

for n in a:
# n war keine Primzahl und wurde schon weggefiltert
if n is None:
        continue

# n ist eine Primzahl
for i in range(2*n, len(a), n):
        a[i] = None

a = list(filter(lambda x: x is not None, a))
print(a)
```

EXERCISE

Implement a function that returns an inner function

- The (outer) function accepts a single parameter
- The inner function uses the parameter passed to the outer function
- The returned value of the inner function depends on the parameter
- Invoke the outer function and invoke the returned function

Create a second program that uses a lambda expression instead of an inner function

EXERCISE

Implement a function that accepts an arbitrary parameter list

- The parameters are functions that accept a number and return a string
- Invoke all passed functions with a random number
- Display the returned value

Call the function with the following parameters

- A single function name (implement this function first)
- A single function name (from above) and a single lambda expression
- 5 lambda expressions