3. Funktionen

Wenn Sie versuchen, alle Anweisungen zur Berechnung einer komplexen Aufgabe in einen einzigen zusammenhängenden Quelltextabschnitt (*Hauptprogramm*) unterzubringen, verliert das Programm schnell an Lesbarkeit oder Sie den Überblick über Ihr eigenes Werk.
Die sog. *Unterprogrammtechnik* bietet hier Abhilfe, um komplexe Problemstellungen in einfach zu beherrschende Teilprobleme zu zerlegen. Ein solches Unterprogramm nennt man in den modernen Programmiersprachen: **Funktion**.

> Eine Funktion ist ein Objekt, das eine bestimmte Teilaufgabe eines Programms lösen kann. Wenn eine Funkton aufgerufen wird, übernimmt sie gewisse Objekte als Eingabe, verabeitet diese und liefert ein Objekt als Ausgabe zurück [2]

[2]: „6. Funktionen | Python 3 - Lernen und professionell anwenden". Zugegriffen: 17. März 2024. [Online]. Verfügbar unter: https://learning.oreilly.com/library/view/python-3/9783958457935/xhtml/ch12.xhtml

Bei der Verwendung von Funktionen sind zwei fundamentale Ideen von Bedeutung:

- schrittweise Verfeinerung
- Rekursion

Python bietet eine Reihe an Standardfunktionen (*built-in functions*):

- `int(x)` : Erzeugt eine neue ganze Zahl aus einem String, byte oder float x.
- `int(string, base)` : Erzeugt eine ganze Zahl aus einer String-Kodierung im Zahlensystem mit der Basis base
- `float(x)` : Erzeugt eine Fließkommazahl aus einem String, byte oder int x
- `str(x)` : Formatiert das Objekt x als Unicode-String. Die Formatierung ist dieselbe wie bei `print(x)`
- `chr(code)` : Erzeugt aus einer ganzzahligen (int) Unicode-Kodierung code ein Zeichen eines einelementigen Unicode-String
- `ord(zeichen)` : Liefert die Kodierung eines einelementigen Unicode-String (↔ `chr(code)`)
- `hex(n), oct(n), bin(n)` : Kodiert eine ganze Zahl n im Hexadezimal-, Oktal- oder Dualsystem. Liefert einen Unicode-String mit Präfix `0x` , `0o` oder `0b`

In [5]:
```
# TODO Beispiele
x = input("Deine Lieblingszahl: ")
print(3*x)
print(type(x))
print(int(x)*3)

chr(0x61)
ord('a')
```

```
Deine Lieblingszahl: 3
333
<class 'str'>
9
```

Out[5]: 97

Hintergrund: Call by what?

Bekannt (aus C/C++):

call by value	call by reference
- der Funktion wird ein Wert übergeben, der zwar von der Funktion verarbeitet wird, jedoch ohne dass es Auswirkungen auf das Originalobjekt hat	- Funktion wird eine Referenz (Zeiger) übergeben. Damit wird das Objekt selber verarbeitet und Änderungen daran bleiben bestehen.

Beides gibt es bei Python nicht mehr, sondern nur noch:

call by object
- es werden mit einer Funktion immer Objekte (oder deren Namen) übergeben. Ob Änderungen an diesen bestehen bleiben, hängt letztendlich vom Typ des Objekts ab. Es gibt veränderbare wie auch unveränderbare Objekte (*mutable* und *immutable*)

3.1 Definition von Funktionen

Die Definition einer Funktion muss folgendem Format entsprechen:

```
def funktionsname (parameterliste):
    anweisungsblock
```

In [10]:
```
# TODO Beispiel einer einfachen Funktion
def sagHallo(name):
    print("Hallo", name)

x = sagHallo("Florian")
print(x)
```

```
Hallo Florian
None
```

In [14]:
```
# Programmbeispiel zur Primzahlermittlung
def primzahl (zahl):
    if zahl <= 1:
        prim = False
    elif zahl == 2:
        prim = True
    else:
        for i in range(2, zahl//2 +1):
            if zahl % i == 0:   # Teiler gefunden
                prim = False
                break
        else: prim = True       # kein Teiler gefunden
    return prim
```

```
primzahl(101073)
```

Out[14]: False

Besser:

In [16]:
```
# Programmbeispiel zur Primzahlermittlung
def primzahl (zahl):
    if zahl <= 1:
        return False
    elif zahl == 2:
        return True
    else:
        for i in range(2, zahl//2 +1):
            if zahl % i == 0:   # Teiler gefunden
                return False
        return True             # kein Teiler gefunden

primzahl(2)
```

Out[16]: True

3.2 Ausführung von Funktionen

3.2.1 Globale und lokale Namen

In [17]:
```
# Beispiel zu "Call by object"
def f(y) :
    print("1. print in Funktion: id(y):",id(y), "y = ", y)
    y = 3
    print("2. print in Funktion: id(y):",id(y), "y = ", y)
#Hauptprogramm
print("Call by Object Reference")
y = 17
print ("1. print im Hauptprogramm: id(y): ", id(y), "y = ", y)
f(y)
print ("2. print im Hauptprogramm: id(y): ", id(y), "y = ", y)
```

```
Call by Object Reference
1. print im Hauptprogramm: id(y):  140721473176872 y =  17
1. print in Funktion: id(y): 140721473176872 y =  17
2. print in Funktion: id(y): 140721473176424 y =  3
2. print im Hauptprogramm: id(y):  140721473176872 y =  17
```

Wieso wird in der letzten print-Ausgabe nicht `y = 3` ausgegeben?

Funktionen verfügen über einen eigenen *Namensraum*. Das bedeutet, dass jede Variable, die man innerhalb einer Funktion definiert, automatisch einen **lokalen** Gültigkeitsbereich hat. Das bedeutet wiederum, dass egal was man mit dieser Variable innerhalb der Funktion anstellt, dies keinen Einfluss auf andere (**globale**) Variablen außerhalb der Funktion hat, auch wenn diese den gleichen Namen haben. Der Funktionsrumpf/Anweisungsblock ist also der Gültigkeitsbereich einer solchen Variablen.

In [20]:
```
# TODO Beispiel für einen häufigen Fehler
# Restarten Sie den Kernel, damit Sie die Fehlermeldung auch sicher sehen
```

```
def eins():
    a = 10
    print("A in Funktion eins(): ",a)
def zwei():
    b = a + 10
    print("B in Funktion zwei() ", b)
a = 1
eins()
zwei()
```

```
A in Funktion eins():  10
B in Funktion zwei()  11
```

Um einen Einblick in die vom Python-Interpreter erzeugten Namensräume zu gewinnen, gibt es die Funktionen `globals()` und `locals()`:

In [1]:
```
# TODO Beispiel ohne Fehler mit Ausgabe der Namensräume
def eins():
    a = 10
    print("A in Funktion eins(): ",a)
    print("Globals von Funktion eins(): ", globals())
    print("Lokalen von Funktion eins(): ", locals())
def zwei():
    b = a + 10
    print("B in Funktion zwei() ", b)
    print("Globals von Funktion zwei(): ", globals())
    print("Lokalen von Funktion zwei(): ", locals())
a = 1
eins()
zwei()
print("Globals vom Hauptprogramm : ", globals())
print("Lokalen vom Hauptprogramm: ", locals())
```

A in Funktion eins(): 10
Globals von Funktion eins(): {'__name__': '__main__', '__doc__': 'Automatically created module for IPython interactive environment', '__package__': None, '__loader__': None, '__spec__': None, '__builtin__': <module 'builtins' (built-in)>, '__builtins__': <module 'builtins' (built-in)>, '_ih': ['', '# TODO Beispiel ohne Fehler mit Ausgabe der Namensräume\ndef eins():\n   a = 10\n   print("A in Funktion eins(): ",a)\n   print("Globals von Funktion eins(): ", globals())\n   print("Lokalen von Funktion eins(): ", locals())\ndef zwei():\n   b = a + 10\n   print("B in Funktion zwei() ", b)\n   print("Globals von Funktion zwei(): ", globals())\n   print("Lokalen von Funktion zwei(): ", locals())\na = 1\neins()\nzwei()\nprint("Globals vom Hauptprogramm : ", globals())\nprint("Lokalen vom Hauptprogramm: ", locals())'], '_oh': {}, '_dh': [WindowsPath('C:/Users/flori/iCloudDrive/Vorlesungen/Python/Workbooks/02')], 'In': ['', '# TODO Beispiel ohne Fehler mit Ausgabe der Namensräume\ndef eins():\n   a = 10\n   print("A in Funktion eins(): ",a)\n   print("Globals von Funktion eins(): ", globals())\n   print("Lokalen von Funktion eins(): ", locals())\ndef zwei():\n   b = a + 10\n   print("B in Funktion zwei() ", b)\n   print("Globals von Funktion zwei(): ", globals())\n   print("Lokalen von Funktion zwei(): ", locals())\na = 1\neins()\nzwei()\nprint("Globals vom Hauptprogramm : ", globals())\nprint("Lokalen vom Hauptprogramm: ", locals())'], 'Out': {}, 'get_ipython': <function get_ipython at 0x000001B9703FACA0>, 'exit': <IPython.core.autocall.ZMQExitAutocall object at 0x000001B97308C210>, 'quit': <IPython.core.autocall.ZMQExitAutocall object at 0x000001B97308C210>, 'open': <function open at 0x000001B971013D80>, '_': '', '__': '', '___': '', 'json': <module 'json' from 'C:\\Users\\flori\\anaconda3\\Lib\\json__init__.py'>, 'getsizeof': <built-in function getsizeof>, 'NamespaceMagics': <class 'IPython.core.magics.namespace.NamespaceMagics'>, '_nms': <IPython.core.magics.namespace.NamespaceMagics object at 0x000001B97306D750>, '_Jupyter': <ipykernel.zmqshell.ZMQInteractiveShell object at 0x000001B97306F750>, 'np': <module 'numpy' from 'C:\\Users\\flori\\anaconda3\\Lib\\site-packages\\numpy__init__.py'>, '_getsizeof': <function _getsizeof at 0x000001B972F9D3A0>, '_getshapeof': <function _getshapeof at 0x000001B9730A5A80>, '_getcontentof': <function _getcontentof at 0x000001B9730A54E0>, 'var_dic_list': <function var_dic_list at 0x000001B9730A5E40>, '_i': '', '_ii': '', '_iii': '', '_i1': '# TODO Beispiel ohne Fehler mit Ausgabe der Namensräume\ndef eins():\n   a = 10\n   print("A in Funktion eins(): ",a)\n   print("Globals von Funktion eins(): ", globals())\n   print("Lokalen von Funktion eins(): ", locals())\ndef zwei():\n   b = a + 10\n   print("B in Funktion zwei() ", b)\n   print("Globals von Funktion zwei(): ", globals())\n   print("Lokalen von Funktion zwei(): ", locals())\na = 1\neins()\nzwei()\nprint("Globals vom Hauptprogramm : ", globals())\nprint("Lokalen vom Hauptprogramm: ", locals())', 'eins': <function eins at 0x000001B9730A56C0>, 'zwei': <function zwei at 0x000001B9730A60C0>, 'a': 1}
Lokalen von Funktion eins(): {'a': 10}
B in Funktion zwei() 11
Globals von Funktion zwei(): {'__name__': '__main__', '__doc__': 'Automatically created module for IPython interactive environment', '__package__': None, '__loader__': None, '__spec__': None, '__builtin__': <module 'builtins' (built-in)>, '__builtins__': <module 'builtins' (built-in)>, '_ih': ['', '# TODO Beispiel ohne Fehler mit Ausgabe der Namensräume\ndef eins():\n   a = 10\n   print("A in Funktion eins(): ",a)\n   print("Globals von Funktion eins(): ", globals())\n   print("Lokalen von Funktion eins(): ", locals())\ndef zwei():\n   b = a + 10\n   print("B in Funktion zwei() ", b)\n   print("Globals von Funktion zwei(): ", globals())\n   print("Lokalen von Funktion zwei(): ", locals())\na = 1\neins()\nzwei()\nprint("Globals vom Hauptprogramm : ", globals())\nprint("Lokalen vom Hauptprogramm: ", locals())'], '_oh': {}, '_dh': [WindowsPath('C:/Users/flori/iCloudDrive/Vorlesungen/Python/Workbooks/02')], 'In': ['', '# TODO Beispiel ohne Fehler mit Ausgabe der Namensräume\ndef eins():\n   a = 10\n   print("A in Funktion eins(): ",a)\n   print("Globals von Funktion eins(): ", globals())\n   print("Lokalen von Funktion eins(): ", locals())\ndef zwei():\n   b = a + 10\n   print("B in Funktion zwei() ", b)\n   print("Globals von Funktion zwei(): ", globals())\n   print("Lokalen von Funktion zwei(): ", locals())\na = 1\neins()\nzwei()\nprint("Globals vom Hauptprogramm : ", globals())\nprint("Lokalen vom Hauptprogramm: ", locals())'], 'Out': {}, 'get_ipython': <function get_ipython at 0x000001B9703FACA0>, 'exit': <IPython.core.autocall.ZMQExitAutocall object at 0x000001B97308C210>, 'quit': <IPython.core.autocall.ZMQExitAutocall object at 0x000001B97308C210>, 'open': <function open at 0x000001B971013D80>, '_': '', '__': '', '___': '', 'json': <module 'json' from 'C:\\Users\\flori\\anaconda3\\Lib\\json__init__.py'>, 'getsizeof': <built-in function getsizeof>, 'NamespaceMagics': <class 'IPython.core.magics.namespace.NamespaceMagics'>, '_nms': <IPython.core.magics.namespace.NamespaceMagics object at 0x000001B97306D750>, '_Jupyter': <ipykernel.zmqshell.ZMQInteractiveShell object at 0x000001B97306F750>, 'np': <module 'numpy' from 'C:\\Users\\flori\\anaconda3\\Lib\\site-packages\\numpy__init__.py'>, '_getsizeof': <function _getsizeof at 0x000001B972F9D3A0>, '_getshapeof': <function _getshapeof at 0x000001B9730A5A80>, '_getcontentof': <function _getcontentof at 0x000001B9730A54E0>, 'var_dic_list': <function var_dic_list at 0x000001B9730A5E40>, '_i': '', '_ii': '', '_iii': '', '_i1': '# TODO Beispiel ohne Fehler mit Ausgabe der Namensräume\ndef eins():\n   a = 10\n   print("A in Funktion eins(): ",a)\n   print("Globals von Funktion eins(): ", globals())\n   print("Lokalen von Funktion eins(): ", locals())\ndef zwei():\n   b = a + 10\n   print("B in Funktion zwei() ", b)\n   print("Globals von Funktion zwei(): ", globals())\n   print("Lokalen von Funktion zwei(): ", locals())\na = 1\neins()\nzwei()\nprint("Globals vom Hauptprogramm : ", globals())\nprint("Lokalen vom Hauptprogramm: ", locals())', 'eins': <function eins at 0x000001B9730A56C0>, 'zwei': <function zwei at 0x000001B9730A60C0>, 'a': 1}

obals vom Hauptprogramm : ", globals())\nprint("Lokalen vom Hauptprogramm: ", loc
als())'], 'Out': {}, 'get_ipython': <function get_ipython at 0x000001B9703FACA0>,
'exit': <IPython.core.autocall.ZMQExitAutocall object at 0x000001B97308C210>, 'qu
it': <IPython.core.autocall.ZMQExitAutocall object at 0x000001B97308C210>, 'ope
n': <function open at 0x000001B971013D80>, '_': '', '__': '', '___': '', 'json':
<module 'json' from 'C:\\Users\\flori\\anaconda3\\Lib\\json__init__.py'>, 'gets
izeof': <built-in function getsizeof>, 'NamespaceMagics': <class 'IPython.core.ma
gics.namespace.NamespaceMagics'>, '_nms': <IPython.core.magics.namespace.Namespac
eMagics object at 0x000001B97306D750>, '_Jupyter': <ipykernel.zmqshell.ZMQInterac
tiveShell object at 0x000001B97306F750>, 'np': <module 'numpy' from 'C:\\Users\\f
lori\\anaconda3\\Lib\\site-packages\\numpy__init__.py'>, '_getsizeof': <functio
n _getsizeof at 0x000001B972F9D3A0>, '_getshapeof': <function _getshapeof at 0x00
0001B9730A5A80>, '_getcontentof': <function _getcontentof at 0x000001B9730A54E0>,
'var_dic_list': <function var_dic_list at 0x000001B9730A5E40>, '_i': '', '_ii':
'', '_iii': '', '_i1': '# TODO Beispiel ohne Fehler mit Ausgabe der Namensräume\n
def eins():\n   a = 10\n   print("A in Funktion eins(): ",a)\n   print("Global
s von Funktion eins(): ", globals())\n   print("Lokalen von Funktion eins(): ",
locals())\ndef zwei():\n   b = a + 10\n   print("B in Funktion zwei() ", b)\n
print("Globals von Funktion zwei(): ", globals())\n   print("Lokalen von Funktio
n zwei(): ", locals())\na = 1\neins()\nzwei()\nprint("Globals vom Hauptprogramm :
", globals())\nprint("Lokalen vom Hauptprogramm: ", locals())', 'eins': <function
eins at 0x000001B9730A56C0>, 'zwei': <function zwei at 0x000001B9730A60C0>, 'a':
1}
Lokalen von Funktion zwei(): {'b': 11}
Globals vom Hauptprogramm : {'__name__': '__main__', '__doc__': 'Automatically c
reated module for IPython interactive environment', '__package__': None, '__loade
r__': None, '__spec__': None, '__builtin__': <module 'builtins' (built-in)>, '__b
uiltins__': <module 'builtins' (built-in)>, '_ih': ['', '# TODO Beispiel ohne Feh
ler mit Ausgabe der Namensräume\ndef eins():\n   a = 10\n   print("A in Funktio
n eins(): ",a)\n   print("Globals von Funktion eins(): ", globals())\n   print
("Lokalen von Funktion eins(): ", locals())\ndef zwei():\n   b = a + 10\n   pri
nt("B in Funktion zwei() ", b)\n   print("Globals von Funktion zwei(): ", global
s())\n   print("Lokalen von Funktion zwei(): ", locals())\na = 1\neins()\nzwei()
\nprint("Globals vom Hauptprogramm : ", globals())\nprint("Lokalen vom Hauptprogr
amm: ", locals())'], '_oh': {}, '_dh': [WindowsPath('C:/Users/flori/iCloudDrive/V
orlesungen/Python/Workbooks/02')], 'In': ['', '# TODO Beispiel ohne Fehler mit Au
sgabe der Namensräume\ndef eins():\n   a = 10\n   print("A in Funktion eins():
",a)\n   print("Globals von Funktion eins(): ", globals())\n   print("Lokalen v
on Funktion eins(): ", locals())\ndef zwei():\n   b = a + 10\n   print("B in Fu
nktion zwei() ", b)\n   print("Globals von Funktion zwei(): ", globals())\n   p
rint("Lokalen von Funktion zwei(): ", locals())\na = 1\neins()\nzwei()\nprint("Gl
obals vom Hauptprogramm : ", globals())\nprint("Lokalen vom Hauptprogramm: ", loc
als())'], 'Out': {}, 'get_ipython': <function get_ipython at 0x000001B9703FACA0>,
'exit': <IPython.core.autocall.ZMQExitAutocall object at 0x000001B97308C210>, 'qu
it': <IPython.core.autocall.ZMQExitAutocall object at 0x000001B97308C210>, 'ope
n': <function open at 0x000001B971013D80>, '_': '', '__': '', '___': '', 'json':
<module 'json' from 'C:\\Users\\flori\\anaconda3\\Lib\\json__init__.py'>, 'gets
izeof': <built-in function getsizeof>, 'NamespaceMagics': <class 'IPython.core.ma
gics.namespace.NamespaceMagics'>, '_nms': <IPython.core.magics.namespace.Namespac
eMagics object at 0x000001B97306D750>, '_Jupyter': <ipykernel.zmqshell.ZMQInterac
tiveShell object at 0x000001B97306F750>, 'np': <module 'numpy' from 'C:\\Users\\f
lori\\anaconda3\\Lib\\site-packages\\numpy__init__.py'>, '_getsizeof': <functio
n _getsizeof at 0x000001B972F9D3A0>, '_getshapeof': <function _getshapeof at 0x00
0001B9730A5A80>, '_getcontentof': <function _getcontentof at 0x000001B9730A54E0>,
'var_dic_list': <function var_dic_list at 0x000001B9730A5E40>, '_i': '', '_ii':
'', '_iii': '', '_i1': '# TODO Beispiel ohne Fehler mit Ausgabe der Namensräume\n
def eins():\n   a = 10\n   print("A in Funktion eins(): ",a)\n   print("Global
s von Funktion eins(): ", globals())\n   print("Lokalen von Funktion eins(): ",
locals())\ndef zwei():\n   b = a + 10\n   print("B in Funktion zwei() ", b)\n
print("Globals von Funktion zwei(): ", globals())\n   print("Lokalen von Funktio

n zwei(): ", locals())\na = 1\neins()\nzwei()\nprint("Globals vom Hauptprogramm :
", globals())\nprint("Lokalen vom Hauptprogramm: ", locals())', 'eins': <function
eins at 0x000001B9730A56C0>, 'zwei': <function zwei at 0x000001B9730A60C0>, 'a':
1}
Lokalen vom Hauptprogramm: {'__name__': '__main__', '__doc__': 'Automatically cr
eated module for IPython interactive environment', '__package__': None, '__loader
__': None, '__spec__': None, '__builtin__': <module 'builtins' (built-in)>, '__bu
iltins__': <module 'builtins' (built-in)>, '_ih': ['', '# TODO Beispiel ohne Fehl
er mit Ausgabe der Namensräume\ndef eins():\n   a = 10\n   print("A in Funktion
eins(): ",a)\n   print("Globals von Funktion eins(): ", globals())\n   print("L
okalen von Funktion eins(): ", locals())\ndef zwei():\n   b = a + 10\n   print
("B in Funktion zwei() ", b)\n   print("Globals von Funktion zwei(): ", globals
())\n   print("Lokalen von Funktion zwei(): ", locals())\na = 1\neins()\nzwei()
\nprint("Globals vom Hauptprogramm : ", globals())\nprint("Lokalen vom Hauptprogr
amm: ", locals())'], '_oh': {}, '_dh': [WindowsPath('C:/Users/flori/iCloudDrive/V
orlesungen/Python/Workbooks/02')], 'In': ['', '# TODO Beispiel ohne Fehler mit Au
sgabe der Namensräume\ndef eins():\n   a = 10\n   print("A in Funktion eins():
",a)\n   print("Globals von Funktion eins(): ", globals())\n   print("Lokalen v
on Funktion eins(): ", locals())\ndef zwei():\n   b = a + 10\n   print("B in Fu
nktion zwei() ", b)\n   print("Globals von Funktion zwei(): ", globals())\n   p
rint("Lokalen von Funktion zwei(): ", locals())\na = 1\neins()\nzwei()\nprint("Gl
obals vom Hauptprogramm : ", globals())\nprint("Lokalen vom Hauptprogramm: ", loc
als())'], 'Out': {}, 'get_ipython': <function get_ipython at 0x000001B9703FACA0>,
'exit': <IPython.core.autocall.ZMQExitAutocall object at 0x000001B97308C210>, 'qu
it': <IPython.core.autocall.ZMQExitAutocall object at 0x000001B97308C210>, 'ope
n': <function open at 0x000001B971013D80>, '_': '', '__': '', '___': '', 'json':
<module 'json' from 'C:\\Users\\flori\\anaconda3\\Lib\\json__init__.py'>, 'gets
izeof': <built-in function getsizeof>, 'NamespaceMagics': <class 'IPython.core.ma
gics.namespace.NamespaceMagics'>, '_nms': <IPython.core.magics.namespace.Namespac
eMagics object at 0x000001B97306D750>, '_Jupyter': <ipykernel.zmqshell.ZMQInterac
tiveShell object at 0x000001B97306F750>, 'np': <module 'numpy' from 'C:\\Users\\f
lori\\anaconda3\\Lib\\site-packages\\numpy__init__.py'>, '_getsizeof': <functio
n _getsizeof at 0x000001B972F9D3A0>, '_getshapeof': <function _getshapeof at 0x00
0001B9730A5A80>, '_getcontentof': <function _getcontentof at 0x000001B9730A54E0>,
'var_dic_list': <function var_dic_list at 0x000001B9730A5E40>, '_i': '', '_ii':
'', '_iii': '', '_i1': '# TODO Beispiel ohne Fehler mit Ausgabe der Namensräume\n
def eins():\n   a = 10\n   print("A in Funktion eins(): ",a)\n   print("Global
s von Funktion eins(): ", globals())\n   print("Lokalen von Funktion eins(): ",
locals())\ndef zwei():\n   b = a + 10\n   print("B in Funktion zwei() ", b)\n
print("Globals von Funktion zwei(): ", globals())\n   print("Lokalen von Funktio
n zwei(): ", locals())\na = 1\neins()\nzwei()\nprint("Globals vom Hauptprogramm :
", globals())\nprint("Lokalen vom Hauptprogramm: ", locals())', 'eins': <function
eins at 0x000001B9730A56C0>, 'zwei': <function zwei at 0x000001B9730A60C0>, 'a':
1}

Die Ausgabe ist leider etwas unübersichtlich, trotzdem lässt sich Folgendes erkennen:

- Beim Hauptprogramm sind lokaler und globaler Namensraum (immer) identisch
- Die globalen Namensräume der Funktionen entsprechen denen des Hauptprogramms
- Die lokalen Namensräume der Funktionen enthalten lediglich die lokal definierten Informationen

3.2.2 Die global-Anweisung - Seiteneffekte

In [3]:
```
# Einfache Verdopplungsfunktion
# TODO Fehler beheben
def verdopple():
```

```
    global x # x wird globale Variable
    x = x*2  # Seiteneffekt
x = 2
verdopple()
x
```

Out[3]: 4

Durch die `global` -Anweisung wird die Variable in den globalen Namensraum eingetragen. Eine Zuweisung wirkt sich damit auf die betreffende Variable des Hauptprogramms aus. Dieses nennt man **Seiteneffekt**.

Sollen mehrere Variablen global sein, schreibt man z.B. `global x, y, z` .

3.2.3 Parameterübergabe

Der Grund, warum die Deklarierung einer globalen Variablen in Python nicht automatisch, sondern explizit stattfinden muss, liegt daran, dass die Benutzung von globalen Variablen generell als schlechter Propgrammierstil betrachtet wird.

Besser ist daher das verwenden von Funktionsparametern und/oder der Rückgabe eines Werts durch die `return` -Anweisung.

Man könnte an dieser Stelle erneut sagen, dass durch das Design von Python gewissermaßen ein guter Programmierstil erzwungen wird.

Man sagt beim Aufruf einer Funktion mit Parametern: Das Argument bzw. der Parameter x wird der Funktion übergeben. Die übergebenen Parameter werden hierbei wie **lokale** Variablen behandelt. Das bedeutet, dass alle Operationen innerhalb der Funktion *keine Auswirkungen* auf den aktuellen Parameter im Hauptprogramm haben.

Ausnahme: Es handelt sich um ein veränderbares (*mutable*) Objekt (z.B. Listen, Sets, Bytearrays)

In [2]:
```
# Einfache Verdopplungsfunktion mit Parameterübergabe
# TODO dafür sorgen, dass es funktioniert
def verdopple(x):
    x = x*2
    return x
x = 2
x = verdopple(x)
x
```

Out[2]: 4

3.3 Voreingestelle Parameterwerte

Für manche Funktionen benötigt man optionale Argumente, die bei Aufruf auch weggelassen werden können, ohne dass eine Fehlermeldung erscheint. Dafür müssen jedoch bestimmte Default-Werte voreingestellt werden:

```
def funktion (arg1=wert1, arg2=wert2, ...):
```

In [5]:
```
# Beispiel zur Berechnung des Umfangs eines Rechtecks
# TODO optionale Parameter
def umfang ( laenge = 2, breite = 1):
    return 2*(laenge + breite)
umfang(5)
```

Out[5]: 12

Wie sieht es aus, wenn ich der Funktion `umfang()` nur die Breite übergeben möchte und für Länge der Default-Wert verwendet werden soll?

3.3.1 Schlüsselwortparameter

Bisher wurden die Argumente in exakt der Reihenfolge, die im Funktionskopf vorgegeben war, übergeben. Man spricht hierbei von *Positionsargumenten*, weil sich die Zuordnung eines Arguments aus der Position in der Argumentliste ergibt.

Positionsargumente sind jedoch eine potenzielle Quelle für semantische Fehler. In komplexen Programmen kann das Vertauschen der Reihenfolge zu unbeschreiblichen Fehlern führen, ohne dass dies zu einer Fehlermeldung des Systems führt.

Deshalb ist es sinnvoll, beim Funktionsaufruf *Schlüsselargumente* (*keyword arguments*) in der Form `Schlüsselwort=Wert` zu verwenden.

In [7]:
```
# Beispiel zur Berechnung des Umfangs eines Rechtecks
# TODO Schlüsselwortparameter verwenden
def umfang ( laenge = 2, breite = 1):
    return 2*(laenge + breite)
umfang(breite = 5, laenge = 2)
umfang(breite = 2)
```

Out[7]: 8

3.3.2 Beliebige Anzahl von Parametern

Häufig hat man den Fall, dass die Anzahl der beim Aufruf nötigen Parameter im Vorhinein nicht bekannt sind. Dafür gibt es in der Informatik folgende wichtige Begriffe:

- *Arität*: Parameteranzahl von Funktionen, Prozeduren oder Methoden
- *variadische Funktion*: Funktionen mit unbestimmter Arität

In Python werden variadische Funktionen mittels des `*` - Operator vor einem Parameter definiert.

In [47]:
```
# TODO Beispiel einer variadischen Funktion
def varfu(*x):
    print(type(x))
varfu(23) # Ausgabe als Tupel
```

```
<class 'tuple'>
```

3.3.3 Beliebige Schlüsselwortparameter

Es gibt auch einen Mechanismus für eine beliebige Anzahl von Schlüsselwortparametern. In Python wird dafür der `**`-Operator vor einen Parameter geschrieben

In [13]:
```
# TODO Beispiel für beliebige Schlüsselwortparameter
def f(**args):
    print(args)
f(de = "German", en = "English", fr = "French") # Ausgabe als Dictionary
```

```
{'de': 'German', 'en': 'English', 'fr': 'French'}
```

In [53]:
```
def f(*arg, **args):
    print(arg, args)
f( ) # Ausgabe als Dictionary
```

```
() {}
```

3.4 Rekursion

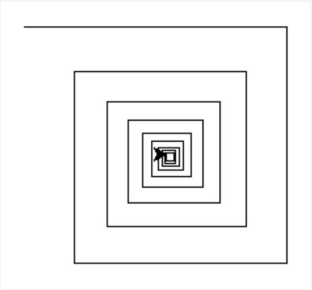
> *Recursion is an order of magnitude more complicated than repitition.* - Dijkstra

3.4.1 Experimente zur Rekursion

In [16]:
```
# Eine rekursive Spirale
from turtle import *

def spirale(x):
    if x < 5:       # Abbruchbedingung (notwendig!)
        return
    else:
        forward(x)
        right(90)
        spirale(x*0.9)
        return


spirale(200)
```

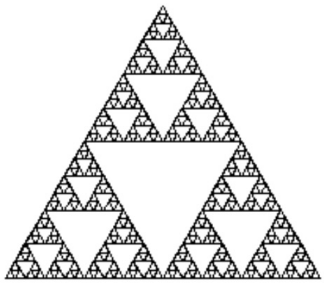


In [20]:
```
# Führen Sie diese Zelle aus, um das Turtle-Fenster zu schließen
from turtle import *
bye()
```

In [19]:
```
# Sierpinski-Dreieck
from turtle import *
def sierpinski(x):
    if x < 5:
        return
    else:
        fd(x)
        right(120)
        fd(x)
        right(120)
        fd(x)
        right(120)
        sierpinski(x/2)
        fd(x/2)
        sierpinski(x/2)
```

```
        back(x/2)
        right(60)
        fd(x/2)
        left(60)
        sierpinski(x/2)
        right(60)
        back(x/2)
        left(60)
        return

speed(0)
left(60)
sierpinski(200)
hideturtle()
```



3.4.2 Rekursive Zahlenfunktionen

Als klassisches Beispiel aus der Mathematik wäre eine rekursive Definition der Fakultät:

```
1! = 1        # Abbruchbedingung
n! = n*(n-1)! # rekursive Anweisung für alle natürlichen Zahlen n > 1
```

Am Beispiel `4!`

```
= 4*3!
= 4*3*2!
= 4*3*2*1!
= 4*3*2*1
= 24
```

In [25]:
```
# TODO Python-Funktion zur Berechnung der Fakultät
def fak(n):
    if n == 1: # Abbruchbedingung
        return 1
    else: # Rekursionsaufruf
        return n*fak(n-1)

fak(1)
```

Out[25]: 1

Wichtig: Eine rekursive Funktion muss eine bedingte Anweisung enthalten, die den Abbruch der Rekursion ermöglicht.
Aber: Die bloße Existenz einer Abbruchbedingung ist natürlich noch keine Garantie, dass diese irgendwann erfüllt sind. Die Folge einer fehlerhaften Abbruchbedingung wäre eine *Endlosrekursion*.

3.4.3 Rekursionstiefe

Die Anzahl der rekursiven Aufrufe nennt man *Rekursionstiefe*. Bei vielen Aufrufen kann also die Rekursionstiefe sehr groß werden, was dazuführen kann, dass der Arbeitsspeicher eines Computers nicht mehr ausreicht.
Der Python-Interpreter beachtet hierbei eine voreingestelle Obergrenze.

In [26]:
```
# Austesten der Rekursionstiefe und ihrer Grenzen
i = 0
def f():
    global i
    i += 1
    f()     # Rekursionsaufruf
f()
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
Cell In[26], line 7
      5     i += 1
      6     f()     # Rekursionsaufruf
----> 7 f()

Cell In[26], line 6, in f()
      4 global i
      5 i += 1
----> 6 f()

Cell In[26], line 6, in f()
      4 global i
      5 i += 1
----> 6 f()

    [... skipping similar frames: f at line 6 (2970 times)]

Cell In[26], line 6, in f()
      4 global i
      5 i += 1
----> 6 f()

RecursionError: maximum recursion depth exceeded
```

In [27]:
```
# Code zum Ermitteln der maximalen Rekursionstiefe
import sys
print("Maximale Rekursionstiefe: ",sys.getrecursionlimit())
```

```
Maximale Rekursionstiefe:  3000
```

Fazit zur Rekursion in Python:

Vorteile	Nachteile
- kurze und elegante Formulierungen für Problemlösungen	- benötigen viel Speicherplatz
- besseres Verständis der Lösung	- arbeiten häufig ineffizient, was sich durch lange Laufzeiten ausdrückt

3.5 Funktionen als Objekte

In der Standard-Typ-Hierarchie von Python werden Funktionen als aufrufbare Objekte (*callable objects*) bezeichnet. Sie werden sozusagen "gleich behandelt" wie Variablen. Funktionen besitzen daher auch eine **Identität**, einen **Typ** und einen **Wert** (siehe auch 2.2.1 Daten als Objekte).

In [28]:
```
# TODO Beispiel an der Standardfunktion len()
print(id(len))
print(type(len))
print(len)
```

```
2265259680096
<class 'builtin_function_or_method'>
<built-in function len>
```

In [30]:
```
# TODO praktisches Beispiel eines Funktionsobjekts
laenge = len
length = len
print(laenge("Wort"))
# Funktionsobjekte können hinsichtlich Gleichheit und Identität verglichen werde
print( len == laenge)
print(len is laenge)
print(len is type)
```

```
4
True
True
False
```

→ kann bei häufig verwendeten, langen Funktionsnamen praktisch sein

Hintergrund: Typen sind *keine* Funktionen

Ein paar der sog. *built-in functions*, wie `int()`, `float()`, `str()`, `bool()`, usw. sind streng genommen **keine** Funktionen, sondern Typen (*typecasting*). Python macht diesen feinsinnigen Unterschied, dennoch sind sie wie Funktionen aufrufbare Objekte (*callable objects*):

In [31]:
```
# Bestimmung des Funktions-"Typen"
print(type(bool))
print(type(str))
print(type(abs))
print(type(len))
```

```
<class 'type'>
<class 'type'>
<class 'builtin_function_or_method'>
<class 'builtin_function_or_method'>
```

3.6 Lambda-Formen

Der lambda-Operator bietet eine Möglichkeit, anonyme Funktionen, also Funktionen ohne Namen, zu schreiben und zu benutzen. Mit der Verwendung von Lambda-Funktionen entfernt man sich von der objektorientieren Programmierung (OOP) und nähert sich der funktionalen Programmierung (FP). FP wird hauptsächlich im technischen und mathematischen Bereich eingesetzt. Python unterstützt und ermöglicht in hohem Maße eine FP, auch wenn Erfinder Guido van Rossum sie am liebsten mit Python 3 wieder entfernt hätte. `lambda`, `map`, `filter` und `reduce` sind Codeerweiterungen der FP. Die Lambda-Funktion hat folgenden Syntax:

```
"lambda" [parameter_list]: expression
```

`lambda` ist hierbei ein Schlüsselwort

In [36]:
```
# TODO Lambda-Beispiele
(lambda x,y: x + y)(2, 3)
addition = lambda x,y: x + y
addition(3,4)
quadratsumme = lambda x,y: x**2 + y**2
quadratsumme(2,3)
kinE = lambda m,v: 0.5*m*v**2
kinE(2, 50)
```

Out[36]:
```
2500.0
```

Benutzung von if-else Anweisungen in Lambda-Funktionen

Syntax:

```
"lambda" [parameter_list]: expression1 if condition else expression2
```

In [40]:
```
# TODO Beispiel
result = lambda x: "Gerade Zahl" if x%2 == 0 else "Ungerade Zahl"
result(91)
```

Out[40]:
```
'Ungerade Zahl'
```

Lambda-Funktionen innerhalb einer Funktion

Syntax:

```
def funktion(y):
    return lambda x: f(x,y) # gibt als Funktionswert eine Lambda-Funktion zurück
```

In [41]:
```
# TODO Beispiel "y-Fach-Funktion"
def yFaches(y):
    return lambda x: x*y

zweifaches = yFaches(2)
dreifaches = yFaches(3)

print(zweifaches(4))
print(dreifaches(4))
```

```
8
12
```

→ wie nützlich die Lambda-Funktionen gerade in Verbindung mit Listen sein können, sehen Sie nächste Woche

3.7 Hinweise zum Programmierstil

3.7.1 Allgemeines

- Iterative Funktionen (mit Schleifen) sind in der Regel rekursiven Funktionen vorzuziehen, weil sie meist weniger Rechenzeit und Arbeitsspeicher benötigen.

3.7.2 Funktionsnamen

Wie bei Variablennamen sollten Funktionsnamen sprechend sein, damit man erkennen kann, was die Funktion leistet. Üblicherweise beginnen diese mit einem kleinen Buchstaben. Man verwendet zudem Verben im Imperativ oder der Funktionsname ist ein Substantiv, das zum Ausdruck bringt, welches Ergebnis die Funktion zurückgibt:

```
# Verben im Imperativ
berechneSumme
getRecursionLimit
anwenden
# Substantive
summe
quadratsumme
min
file
globals
locals
```

3.7.3 Kommentierte Parameter

Zu einer professionellen Dokumentation gehört, dass Sie im Funktionskopf die einzelnen Parameter kommentieren und erklären. Sie schreiben dabei jeden Parameter in verschiedene physische Zeilen, der Python-Interpreter sieht diese dann als eine einzige logische Zeile an

In [42]:
```
def druckeEtikett(  name,           # chemische Bezeichnung (String)
                    formel,         # chemische Formel (String)
                    r_saetze,       # Tupel von Nummern
                    s_saetze,       # Tupel von Nummern
                    gefahrenhinweis,# z.B. "aetzend" (String)
                    fuellmenge      # Füllmenge in g (Integer)
                    ):
    print(name, formel, r_saetze, s_saetze, gefahrenhinweis, fuellmenge)

druckeEtikett("Salzsäure", "HCl", (1,2,3), (4,5,6), "aetzend", 200)
```

```
Salzsäure HCl (1, 2, 3) (4, 5, 6) aetzend 200
```

3.7.4 Docstrings

Ein Docstring wird direkt unter den Funktionskopf eingefügt und in dreifache Anführungszeichen `"""` gesetzt. In der ersten Zeile wird die Aufgabe der Funktion beschrieben, die zweite Zeile bleibt leer und die folgenden Zeilen können Angaben zu folgenden Punkten enthalten:

- Vorbedingungen: Welche Eigenschaften müssen die übergebenen Parameter besitzen?
- Nachbedingungen: Welche Objekte gibt die Funktion zurück?
- Welche globalen Variablen werden verwendet? Welche Seiteneffekte werden verursacht?
- Name des Autors und Datum der letzten Änderung

In [44]:
```python
# Beispiel zur Verwendung eines Docstrings
def tueNichts():
    """ Diese Funktion macht nichts

    Sie verwendet keine Parameter,
    hat keine Seiteneffekte und
    gibt nichts zurück
    F. Hillitzer 18.03.2024
    """
    pass
```

Der Docstring einer Funktion kann mit der `help()`-Funktion zum Vorschein gebracht werden:

In [45]:
```python
help(tueNichts)
```

```
Help on function tueNichts in module __main__:

tueNichts()
    Diese Funktion macht nichts

    Sie verwendet keine Parameter,
    hat keine Seiteneffekte und
    gibt nichts zurück
    F. Hillitzer 18.03.2024
```