Einstieg ins Programmieren
mit Python

Merk-Steine auf dem Weg zu Python für Kids

Autor: Gregor Lingl

Email: glingl@aon.at

Warum Python?

- **Leicht zu lernen. Python-Code ist stets wohl strukturiert und gut lesbar.**
- **Interpretersprache mit interaktiver Shell, die die Erforschung der Sprache ermöglicht.**
- **Python ist eine moderne Sprache, von Grund auf objektorientiert - und ermöglicht doch problemlos in unterschiedlichsten Programmierstilen zu arbeiten.**

Überblick

- **Interaktive ”Shell"**
- **Rechnen: Zahlen und Operationen**
- **Schreiben: Ausgabe mit der `print`-Anweisung**
- **Benennen: Variable sind Namen, die auf Dinge verweisen**
- **Lesen: Eingabe von Daten über die Tastatur (`raw_input`)**
- **Zeichnen mit Turtle-Grafik**
- **Lernen neuer Wörter: Definition von Funktionen**
- **Würfeln: Zufallsgenerator**
- **Entscheiden: Bedingte Anweisungen und Verzweigungen**
- **Wiederholen: Schleifen**
- **Funktionen mit Wert**

IDLE und der IPI

IPI = “interaktiver Python-Interpreter”: Du gibts ein – Python gibt aus …

```
Python 2.2.2 (#37, Oct 14 2002, 17:02:34)
[MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license"
for more information.
IDLE 0.8 -- press F1 for help
>>> print "Viel Spaß mit Python!"
Viel Spaß mit Python!
>>> 2 ** 100
1267650600228229401496703205376L
>>> |
```

Python Shell — File Edit Debug Windows Help

Deine Eingabe: Eine Python-Anweisung

Der IPI führt die Anweisung aus ... ... hier das Ergebnis!

Deine Eingabe: Ein Python-Ausdruck

Der IPI wertet den Ausdruck aus ... ... und schreibt das Ergebnis an.

Eingabe-Prompt

Cursor (Schreibmarke)

IDLE hat auch Editor-Fenster

Editor – Fenster. Damit schreibst du deine Programme

Programm-Ausführung mit Edit|Run Script

Ergebnis der Programmausführung

Rechnen(1)

Python kann rechnen: Der IPI wertet arithmetische Ausdrücke aus und schreibt die Ergebnisse an:

```
>>> 1 + 1
2
>>> 3 * 4
12
>>> (13 + 4) * 3
51
>>> 29 / 8
3.625
>>> 29 // 8     # Ganzzahl-Division
3
>>> 29 % 8      # Rest bei der Division
5
>>> 1 + 1.0
2.0
```

Angaben ganzzahlig ➔ Ergebnis ganzzahlig …

… außer bei neuer Division

Kommazahl unter den Angaben ➔ Ergebnis Kommazahl

(höheres) Rechnen(2)

Python hat ein Modul für mathematische Funktionen:

```
>>> from math import sqrt
>>> sqrt(2)
1.4142135623730951
```

Die Quadratwurzel-Funktion `sqrt` muss aus dem Modul `math` `import`iert werden.

```
>>> 2 ** 10      # Potenzieren
1024             # 1 kilo-
>>> 200.0 ** 10
1.024e+023
```

Exponentialdarstellung für Kommazahlen: $1.024 * 10^{23}$

```
>>> 201 ** 10
107636749520976961802001L
>>> print 201 ** 10
107636749520976961802001
```

`L` für die „interne Darstellung" langer Ganzzahlen

Python kann Ganzzahlarithmetik mit unbeschränkter Genauigkeit!

Schreiben(1)

Python schreibt mit der `print` – Anweisung:

```
>>> print "Hello world!"
Hello world!
```

`print` schreibt hier eine …

```
>>> print "Hi!", "Fein, dass du da bist."
Hi! Fein, dass du da bist.
```

… und hier zwei Zeichenketten.

```
>>> print "3.4*12 =", 3.4*12
3.4*12 = 40.8
```

Der arithmetische Ausdruck wird vom IPI ausgewertet und das Ergebnis wird mit der `print` – Anweisung ausgegeben.

Schreiben(2)

Der IPI schreibt *anders* als die `print` – Anweisung:

```
>>> print "Schon möglich!"
Schon möglich!
>>> "Schon möglich!"
'Schon m\xf6glich!'
>>> print 3.1
3.1
>>> 3.1
3.1000000000000001

>>> "dreikommavier mal zwölf =", 3.4*12
('dreikommavier mal zw\xf6lf =', 40.799999999999997)
```

Interne Darstellung der Zeichenkette

Interne Darstellung der Kommazahl

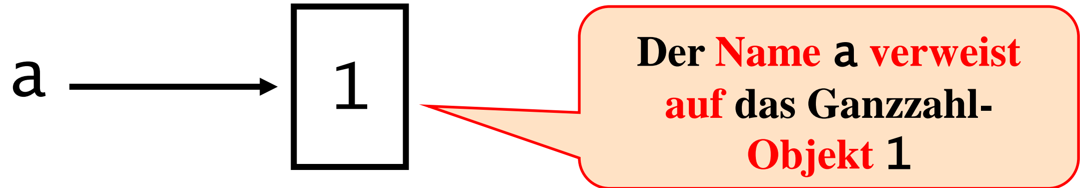
Interne Darstellung der Zeichenkette

Interne Darstellung der Kommazahl

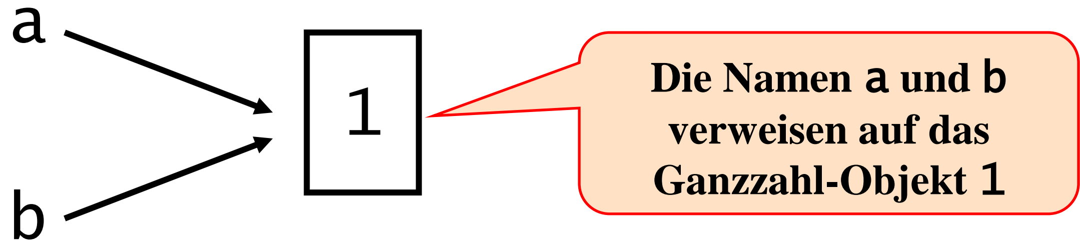
Benennen(1)

Name + Ding = Variable

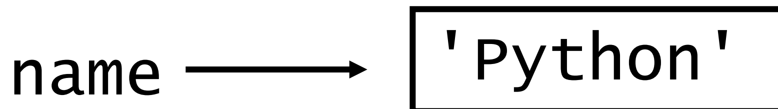
```
>>> a = 1
>>> a
1
```



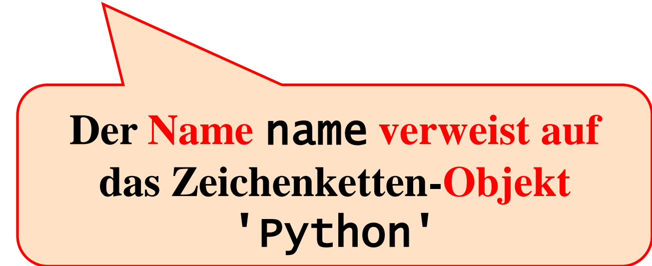

```
>>> b = a
>>> b
1
```




```
>>> name="Python"
>>> name
'Python'
```



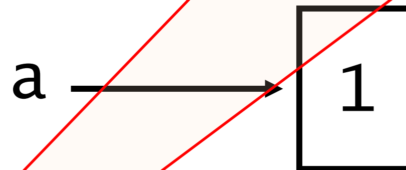

```
>>> print name, "rocks!"
Python rocks!
```



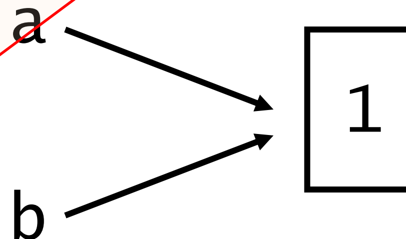
Benennen(2)

Wertzuweisung - **neuerliche Wertzuweisung**

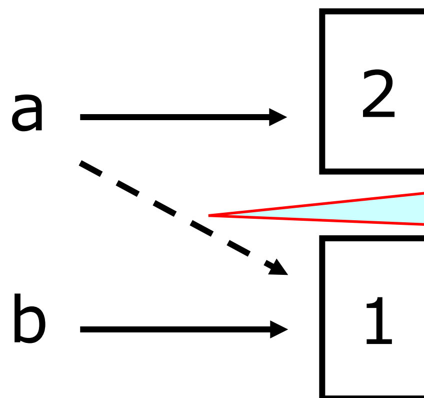
```
>>> a = 1
>>> a
1
```




```
>>> b = a
>>> b
1
```




```
>>> a = a + 1
>>> a
2
>>> b
1
```



Ein neues Ganzzahl-Objekt wird durch die Addition (1+1) erzeugt.

Der alte Verweis wird durch die neue Wertzuweisung (a = ...) zerstört.

Lesen(1)

Die Funktion `raw_input` liest Zeichenketten von der Tastatur ein:

```
>>> name = raw_input()
Harry
>>> name
'Harry'

>>> name = raw_input("Wie heißt du? ")
Wie heißt du? Clara
>>> name
'Clara‘

>>> print "Hallo " + name + ", wie geht's?"
Hallo Clara, wie geht's?
```

Benutzereingabe

„Prompt“

Verkettung von Zeichenketten

Lesen(2)

Einlesen von Zahlenwerten von der Tastatur …

```
>>> alter = raw_input("Wie alt bist du? ")
Wie alt bist du? 15
>>> alter
'15'    # Ein String

>>> jahre = int(alter)
>>> jahre
15

>>> jahre = float(alter)
>>> jahre
15.0
```

… in zwei Schritten:

1. Einlesen eines Strings
2. Umwandeln in den gewünschte Zahlentyp: `int` …

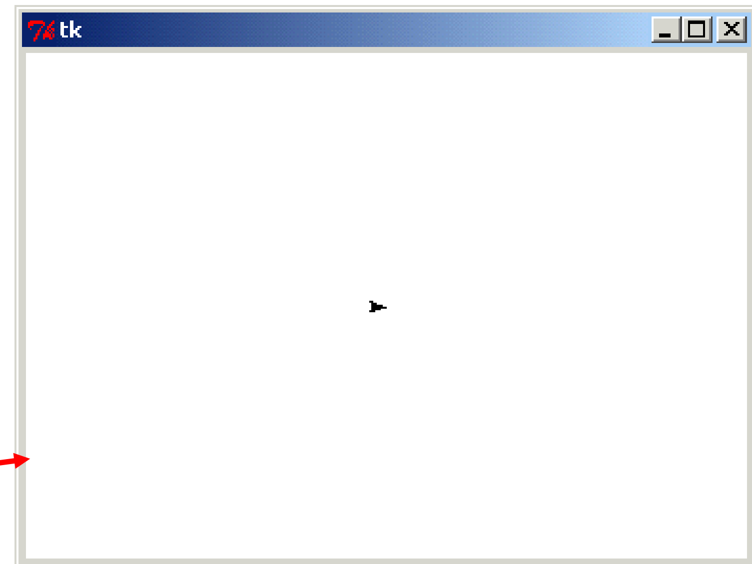
… oder `float`.

Zeichnen(1)

Turtle-Grafik … - ein Software-Tierchen zum Zeichnen:

* `import`iert alle Funktionen aus dem Modul `turtle`!

```
>>> from turtle import *
```

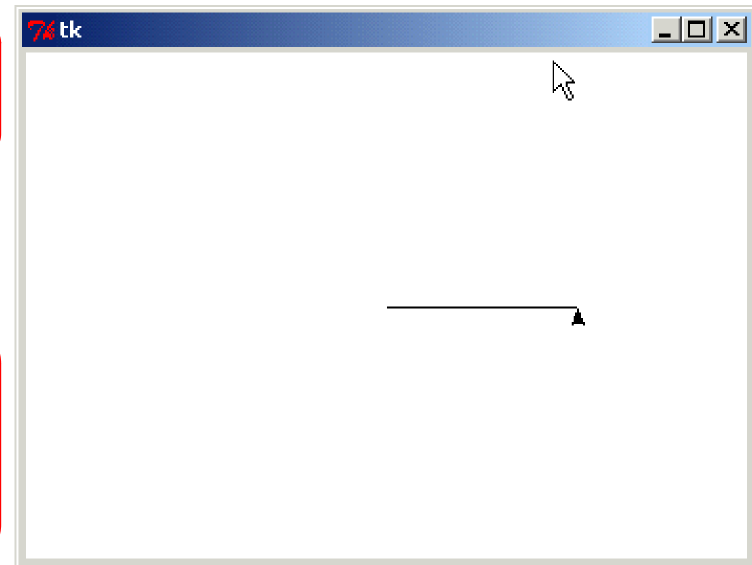



```
>>> reset()
```

Erzeugt Grafik-Fenster, Turtle „schaut nach rechts“.

```
>>> forward(100)
>>> left(90)
```

Turtle geht um 100 Einheiten nach vor und dreht sich um 90° nach links.
Jetzt „schaut sie nach oben“



Zeichnen(2)

… und mehr Turtle-Grafik:

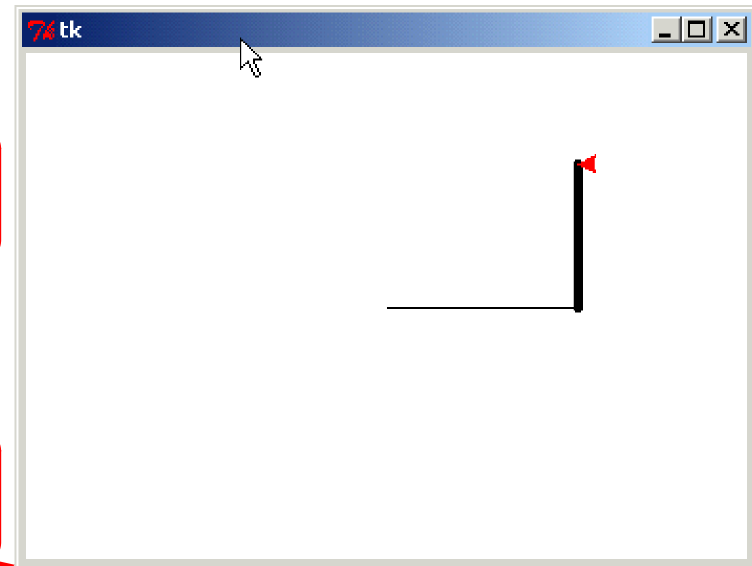
Stellt Strichdicke auf 5 Pixel.

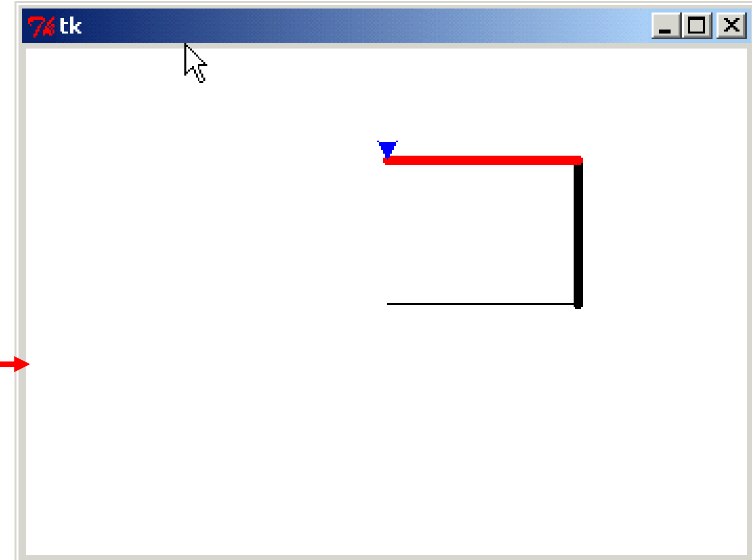
```
>>> width(5)
>>> forward(75)
```

Stellt Strichfarbe auf Rot.

```
>>> left(90)
>>> color("red")

>>> forward(100)
>>> left(90)
>>> color("blue")
```





Zeichnen(3)

Beispiel: gleichseitiges Dreieck

```
>>> reset()
>>> up()
>>> backward(70)
>>> down()
>>> width(7)
>>> color("blue")
>>> fill(1)
>>> forward(140)
>>> left(120)
>>> forward(140)
>>> left(120)
>>> forward(140)
>>> left(120)
>>> color("red")
>>> fill(0)
>>>
```

Zeichenstift anheben … (= Zeichnen ausschalten)

… und wieder absenken.

Füllen einschalten

Zeichnet Dreieck

Füllfarbe wählen

Füllen durchführen und danach Füllen ausschalten.

Zeichnen(4)

Die wichtigsten Turtle-Grafik-Funktionen:

`from turtle import *` . . . importiert alle Funktionen aus dem Modul turtle

`reset()` . . . setzt Grafikfenster auf den Anfangszustand: Turtle in der Fenstermitte, Koordinaten: (0 / 0)

`forward(`*laenge*`)` . . . Turtle geht *laenge* vorwärts

`left(`*winkel*`)` . . . dreht Turtle um *winkel* (in Grad) nach links

`right(`*winkel*`)` . . . dreht Turtle um *winkel* (in Grad) nach rechts

`up()` . . . hebt den Zeichenstift an (schaltet Zeichnen aus!)

`down()` . . . senkt den Zeichenstift ab (schaltet Zeichnen ein!)

Zeichnen(5)

Weitere Turtle-Grafik-Funktionen:

`clear()`	. . . löscht Zeichnungen im Grafikfenster Ort, Farbe, Strichdicke der Turtle bleiben erhalten
`backward(`*laenge*`)`	. . . Turtle geht *laenge* rückwärts
`width(`*breite*`)`	. . . stellt Strichdicke ein (*breite*: Ganzzahl)
`color(`*farbe*`)`	. . . stellt Strichfarbe ein (*farbe*: Farbbezeichnung als Zeichenkette, z. B.: `"red"`, `"black"`)
`tracer(1)`	. . . Schaltet langsames Zeichnen ein (Standard)
`tracer(0)`	. . . Schaltet langsames Zeichnen aus
`fill(1)`	. . . Schaltet Füllen ein. Der im Folgenden von der Turtle ausgeführte Streckenzug wird schließlich durch …
`fill(0)`	. . . mit der aktuellen Farbe gefüllt.

Mit der Definition einer Funktion …

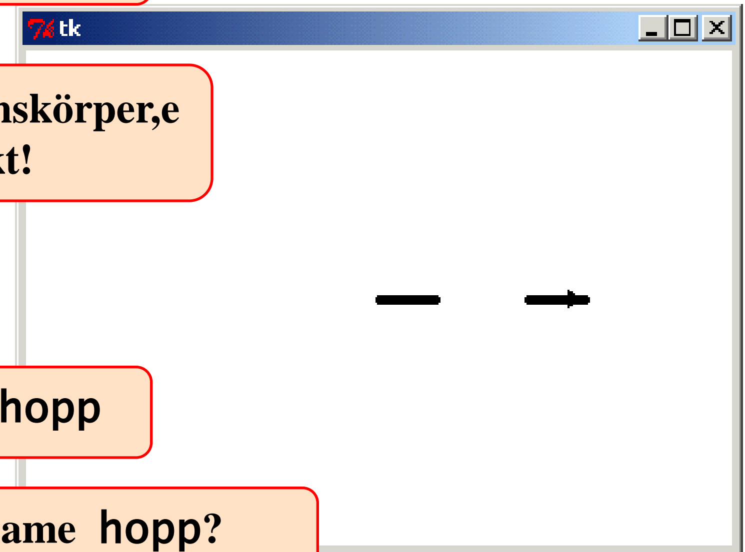
…lernt Python ein neues Wort: hopp

Definition der Funktion hopp

```
>>> def hopp():
        up()
        forward(50)
        down()
```

Funktionskopf

Funktionskörper,e ingerückt!




```
>>> forward(30)
>>> hopp()
>>> forward(30)
>>> hopp
<function hopp at 0x009394A8>
>>>
```

Aufruf der Funktion hopp

Worauf verweist der Name hopp?

Auf ein Funktions-Objekt!

Lernen(2)

Definition einer Funktion in einem Programm (Skript):

Python Shell

File Edit Debug

```
Python 2.
) [MSC 32
Type "cop
 for more
IDLE 0.8
>>>
```

dreieck.py - C:/Py4Kids/dreieck.py

File Edit Windows Help

```
# Skript mit Funktionsdefinition

from turtle import *

def dreieck():
    forward(90)
    left(120)
    forward(90)
    left(120)
    forward(90)
    left(120)

width(7)
color("orange")
dreieck()
```

tk

Editor Fenster

Ergebnis der Ausführung des Skripts dreieck.py

Lernen(3)

Eine Definition einer Funktion *mit Parameter* (Skript):

Python Shell

File Edit

Pyth
) [M
Type
  for
IDLE
>>>

zweidreiecke.py - C:/Py4Kids/zweidreiecke.py

File Edit Windows Help

```
# Funktion mit Parameter
from turtle import *

def dreieck(seite):
    forward(seite)
    left(120)
    forward(seite)
    left(120)
    forward(seite)
    left(120)

width(7)
color("red")
dreieck(100)
left(150)
dreieck(140)
```

Parameter: `seite`

Argument: 100

Ergebnis der Ausführung des Skripts `zweidreiecke.py`

Lernen(4)

Namen in Funktionen (lokale Variable):

Python Shell

File Edit Debug Windows Help

```
>>> zahl = 5
>>> def namen_test(z):
        print "(1) z:", z
        z = 2 * z
        print "(2) z:", z
        zahl = 3 * z
        print "(3) zahl:", zahl


>>> namen_test(zahl)
(1) z: 5
(2) z: 10
(3) zahl: 30
>>> zahl
5
>>> z
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in ?
    z
NameError: name 'z' is not defined
```

Ln: 61 Col: 0

Diese Namen `z` und `zahl` sind nur innerhalb von `namen_test` bekannt: `z` und `zahl` sind *lokale Variable* der Funktion `namen_test`.

Lokale Variable `z`

Lokale Variable `zahl`

Globale Variable `zahl`

Es gibt keine globale Variable `z`, daher Fehlermeldung: `NameError`

Zufallsgenerator:

```
IDLE 0.8 -- press F1
>>>
6 3 2 2 3 6 6 2 3 5
2 5 3 4 3 4 4 3 5 1
3 4 1 1 3 1 3 1 6 6
6 4 6 1 1 5 6 2 2 6
4 4 4 2 1 2 1 4 3 3
5 4 5 5 6 1 6 1 2 5
1 3 3 6 6 2 1 5 5 5
5 5 3 6 2 4 1 4 6 1
2 6 5 1 4 3 3 6 5 3
4 2 4 3 3 1 3 1 2 3
```

zufall.py - C:/Py4Kids/zufall.py

```
# Zufallsgenerator ausprobieren:
from random import randrange

print randrange(1,7),
print randrange(1,7),
print randrange(1,7),
print randrange(1,7),
print randrange(1,7),
print randrange(1,7),
print randrange(1,7),
print randrange(1,7),
print randrange(1,7),
print randrange(1,7),
print
```

Die Funktion `randrange` steht im Modul `random`

`randrange` ist ein „Zufallsgenerator“

Skript `zufall.py` 10 Mal ausgeführt.

`randrange(1,7)` funktioniert wie ein Würfel: erzeugt zufällig Zahlen von 1 bis 6.

Entscheiden(1)

Bedingte Anweisung: `if`

```
*Python Shell*
File  Edit  Debug  Windows
>>> spiel()
1 3
>>> spiel()
2 2 Bingo!
**********
>>> spiel()
2 3
>>> spiel()
5 5 Bingo!
**********
>>> spiel()
1 6
>>>
```

```
spiel.py - C:/Py4Kids/spiel.py
File  Edit  Windows  Help
# Zufallsgenerator ausprobieren:
from random import randrange

def spiel():
    a = randrange(1,7)
    b = randrange(1,7)
    print a, b,
    if a == b:
        print "Bingo!"
        print 10 * "*",
    print
```

Bedingung: `a == b`

Anweisungskopf

Anweisungsskörper, ein eingerückter Block!

Funktion `spiel.py` 5 Mal aufgerufen.

Der Anweisungskörper der `if`-Anweisung wird nur ausgeführt, wenn die *Bedingung* wahr ist.

Entscheiden(2)

Verzweigungsanweisung: `if - else`

```
>>> spiel()
5 6  - leider nicht!
>>> spiel()
6 6 Bingo!
**********
>>> spiel()
1 1 Bingo!
**********
>>> spiel()
4 4 Bingo!
**********
>>> spiel()
5 1  - leider nicht!
>>>
```

spiel2.py - C:/Py4Kids/spiel2.py

```
# Zufallsgenerator ausprobieren:
from random import randrange

def spiel():
    a = randrange(1,7)
    b = randrange(1,7)
    print a, b
    if a == b:
        print "Bingo!"
        print 10 * "*"
    else:
        print " - leider nicht!"
```

Bedingung: `a == b`

Anweisungskopf

`if` - Zweig

`else` - Zweig

Funktion `spiel.py` 5 Mal aufgerufen.

Der `if`-Zweig der Verzweigungsanweisung wird ausgeführt, wenn die *Bedingung* wahr ist, andernfalls wird der `else` – Zweig ausgeführt.

Entscheiden(3)

else+if = elif

Mehrfache Verzweigung: `if - elif - else`

Python Shell

File Edit Debug Windows Help

```
>>> vergleich()
6 größer 2
>>> vergleich()
1 kleiner 3
>>> vergleich()
5 gleich 5
>>> vergleich()
3 kleiner 5
>>> vergleich()
2 kleiner 6
>>> vergleich()
4 größer 3
>>>
```

vergleich.py - C:/Py4Kids/vergleich.py

File Edit Windows Help

```
from random import randrange

def vergleich():
    a = randrange(1,7)
    b = randrange(1,7)
    print a,
    if a == b:
        print "gleich",
    elif a < b:
        print "kleiner",
    else:
        print "größer",
    print b
```

Bedingung 1: `a == b`

Fall 1

Bedingung 2: `a < b`

Fall 2
...

... letzter Fall

Funktion `vergleich.py`
6 Mal aufgerufen.

- * Die Bedingungen werden der Reihe nach getestet.
- * Sobald eine wahr ist, wird der zugehörige Zweig ausgeführt
- * Ist keine Bedingung wahr, wird der `else` – Zweig ausgeführt
- * In jedem Fall wird nur ein Zweig ausgeführt.

Wiederholen (1)

Die Funktion `range` mit einem Argument:

```
>>> range(5)
[0, 1, 2, 3, 4]
```

Alle Argumente müssen ganzzahlig sein!

Die Funktion `range` mit zwei Argumenten:

```
>>> range(6, 9)
[6, 7, 8]
```

Funktionsaufruf

Ergebnis: eine Liste ganzer Zahlen

Die Funktion `range` mit drei Argumenten:

```
>>> range(5, 16, 3)
[5, 8, 11, 14]
```

```
>>> range(5, 16, 3
        range([start,] stop[, step]) -> list of integers
```

Wiederholen (2)

Einfache Zählschleifen:

(ohne Verwendung der Zählvariablen)

```
>>> for i in range(7):
        print '*',
```

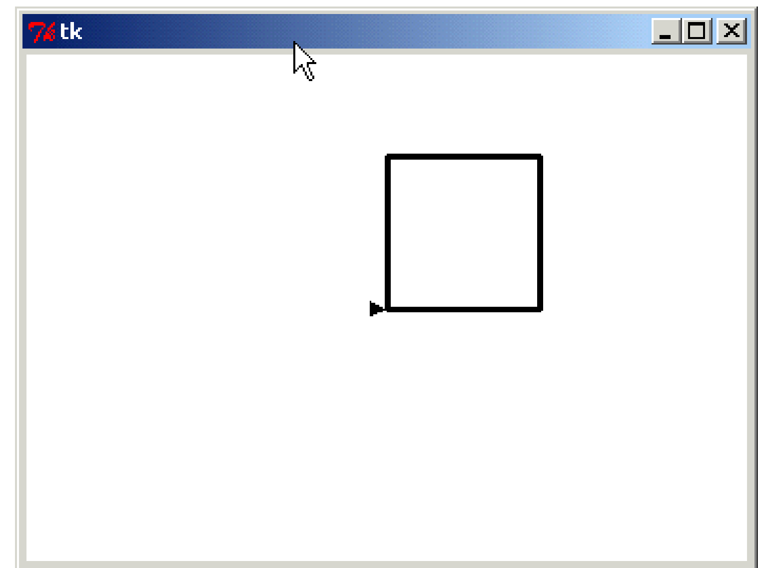
Schleifenkopf

Schleifenkörper, hier nur eine Anweisung, muss eingerückt werden!

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

```
>>> for k in range(4):
        forward(80)
        left(90)
```

Schleifenkörper, hier aus zwei Anweisungen, muss eingerückt werden!



Wiederholen (3)

Zählschleifen mit Verwendung der Zählvariablen:

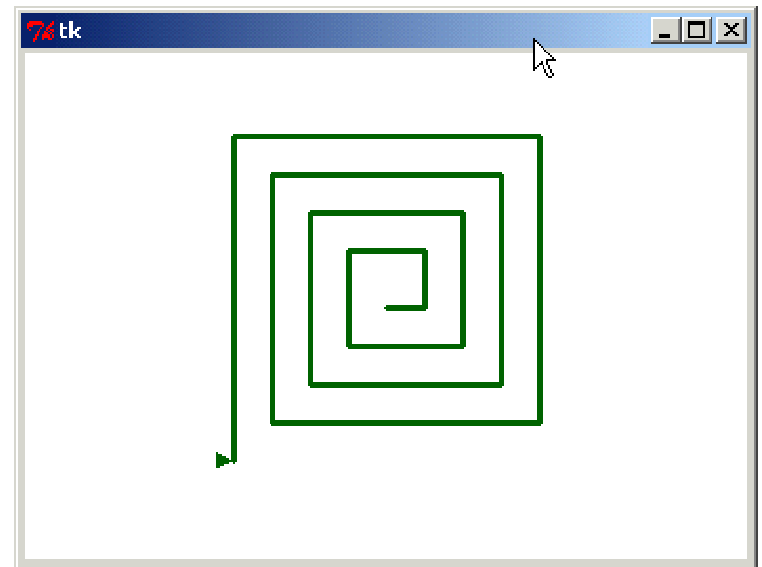
```
>>> for i in range(11):
        print i**2,

0 1 4 9 16 25 36 49 64 81 100
>>> for l in range(20, 180, 10):
        forward(l)
        left(90)
```

Zählvariable: i

Zählvariable: l

Die Zählvariable kommt im Schleifenkörper auch vor!



Wiederholen (4)

Bedingte Schleifen:

```
>>> i = 10
>>> while i > 0:
        print i,
        i = i - 1

10 9 8 7 6 5 4 3 2 1
```

Schleifenvariable: `i`, muss vor Beginn der Schleife initialisiert werden! Hier bekommt sie den Anfangswert 10.

Solange diese Bedingung wahr ist, wird der Schleifenkörper wiederholt.

Schleifenvariable muss im Schleifenkörper verändert werden, damit die Schleife „abbricht“ (d. h. ein Ende findet).

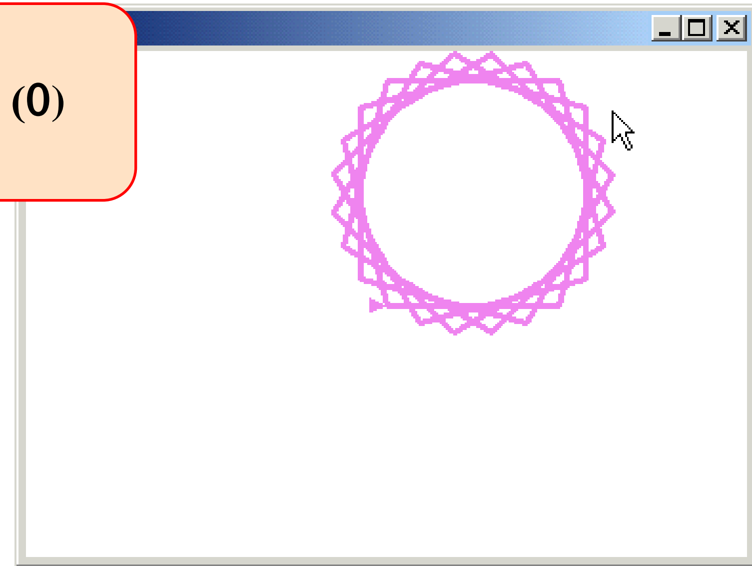
Wiederholen (5)

Bedingte Schleifen:

```
>>> winkel = 0
>>> fertig = 0
>>> while not fertig:
        forward(90)
        left(75)
        winkel = winkel + 75
        fertig = (winkel % 360 == 0)

>>> print winkel
1800
```

`fertig`: Boole'sche Variable, mit „Falsch" (0) initialisiert.



Bedingung: Winkel ist ein Vielfaches von 360°. Sobald diese Bedingung – und damit `fertig` – „wahr" wird, wird `not fertig` „falsch". Daher bricht die Schleife ab.

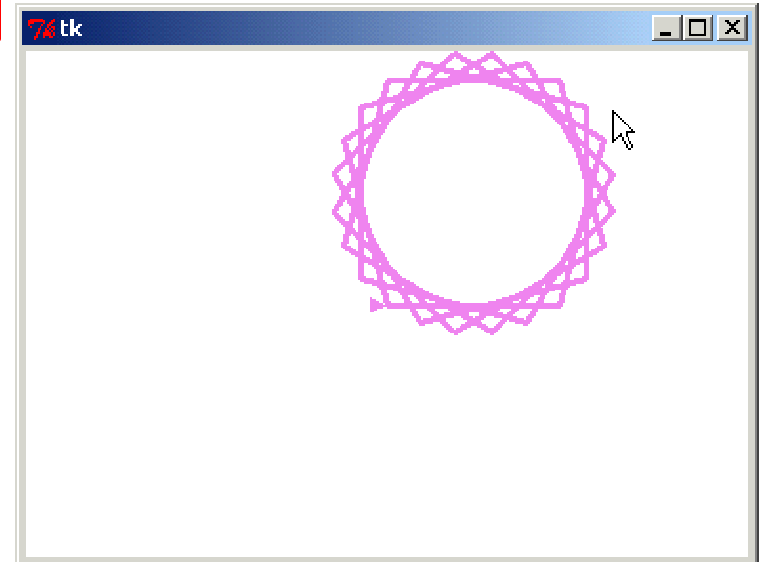
Wiederholen (6)

Alternativ: bedingte Schleife mit `break`:

```
>>> winkel = 0
>>> while 1:
        forward(90)
        left(75)
        winkel = winkel + 75
        if winkel % 360 == 0:
            break

>>> print winkel
1800
```

Endlos? (1 ist „wahr“)



Nein!
`break` **bricht die** (innerste) **Schleife ab.**

Funktionen mit Wert(1)

Die `return` – Anweisung

```
>>> def durchschnitt(a, b):
        d = (a + b) / 2.0
        return d
```

Die `return` – Anweisung gibt das Objekt `d` (hier eine Zahl) zurück.

Ein zurückgegebenes Objekt kann…

```
>>> durchschnitt(1, 4)
2.5
```

…vom IPI angeschrieben werden

```
>>> print durchschnitt(2.62, 7.88)
5.25
```

…von `print` ausgegeben werden

```
>>> durchschnitt(2, 7) + durchschnitt(5, 3)
8.5
```

… in arithmetischen Ausdrücken verwendet werden

Funktionen mit Wert(2)

Beispiel: eine Funktion, die eine Zeichenkette zurück gibt:

```
>>> def aufteilung(anzahl):
        haufen = anzahl // 5
        rest = anzahl % 5
        s = haufen * "IIIII " + rest * "I"
        return s

>>> print aufteilung(18)
IIIII IIIII IIIII III
>>> print aufteilung(6)
IIIII I
```

Funktionen mit Wert(3)

Beispiel: die Faktorielle - Funktion:

„n – Faktorielle“ ist folgendermaßen definiert: 0! = 1 und n! = Produkt aller ganzen Zahlen von 1 bis n für n > 0.

```
>>> def fak(n):
        f = 1
        for k in range(2, n+1):
                f =  f * k
        return f

>>> for n in range(10):
        print fak(n),

1 1 2 6 24 120 720 5040 40320 362880
>>> print fak(66)
544344939077443064003729240247842752644293064388798874532860126869671081148416000000000000000
```