



Funktionen



- Umfangreiche Programme werden in Funktionen aufgeteilt, dadurch werden sie modularer und einfacher zu warten.
- Funktionen werden mit der def-Anweisung definiert

```
def add( x, y ):  
    return x+y
```

- Funktionsaufrufe erfolgen durch Angabe des Funktionsnamens und der Funktionsargumente

```
a = add( 3, 5 )
```

- Anzahl der Argumente muß mit der Funktionsdefinition übereinstimmen, sonst wird ein Type-Error ausgelöst



Achtung



- Kein Rückgabebetyp deklariert!
 - Funktion kann Objekte von verschiedenem Typ jedesmal liefern!
 - Große Flexibilität, große Gefahr
- Parameter haben keinen Typ deklariert!
 - Dito



Parameterübergabe



- Ist in Python prinzipiell *Call-by-Reference*
 - Funktion bekommt Referenz (Zeiger), keine Kopie des Wertes
 - D.h., Parameter können in der Funktion geändert werden!

```
def set( x ):  
    x[0] = 3  
  
a = [ 1, 2, 3 ]  
print a  
set( a )  
print a
```

Ausgabe

```
123  
323
```

- Ausnahmen: "einfache" Datentypen (Integer, Float, String)

```
def set( x ):  
    x = 3  
  
a = 1  
print a  
set( a )  
print a
```

Ausgabe

```
1  
1
```



Keyword-Parameter



- Die Parameter in einem "klassischen" Aufruf der Art

```
foo( 1.0, "hello", [1,2,3] )
```

heißen "*positional arguments*", weil ihre Zuordnung zu den formalen Parametern durch die Position in der Liste aller Argumente gegeben ist

- Alternative: Parameter-Übergabe durch *Key-Value-Paare*:

```
def new_frame(text, name, bg_color, fg_color, font, size):  
    ...whatever...  
  
new_frame("Hello World", name="hello", font="Helvetica",  
          size=4, bg_color="blue", fg_color="red" )
```



Default-Werte

- Angabe aller Parameter bei langer Parameterliste manchmal mühsam
- Lösung: **Default-Argumente**

```
def new_frame(text, name="upper_left", bg_color="white",
              fgcolor="black", font="Ariel", size=2):
    ..whatever...

new_frame("Hello World", font="Helvetica")
new_frame("Our products", "index_frame", size=4,
          bg_color="blue")
```

- Alle Parameter, die im Aufruf **nicht** angegeben werden (*positional* oder *key/value*), werden mit Default-Argumenten belegt



- Regeln für Argumente
- Beispiel-Funktion:

```
def f(name, age=30):
    ..whatever..
```

- Argument darf nicht sowohl *positional* als auch als *Key/Value* gegeben werden:

```
f("aaron", name="sam") --> ValueError
```

- *Positional* Argumente müssen *Key/Value*-Argumenten voranstehen

```
f(name="aaron", 34) --> SyntaxError
```

- Alle Argumente ohne Default-Werte müssen definiert werden

```
f() --> ValueError
```



Funktionen als Parameter

- Funktionen sind vollwertige Objekte
- Funktionen können auch Funktionen als Parameter erhalten
 - Analog in C: Funktionszeiger; analog in C++: Funktoren
- Beispiel: Sortieren von Listen

```
list.sort( cmpfunc )
```



Beispiel `map`

- Die Funktion `t = map(func, s)` wendet die Funktion `func()` auf alle Elemente einer Liste `s` an und gibt eine neue Liste `t` zurück

```
a = [1, 2, 3, 4, 5, 6]
def triple(x):
    return 3*x
b = map( triple, a ) # b = [3,6,9,12,15,18]
```

- Weiteres `map`-Beispiel: alle *command line arguments* als `int`'s lesen

```
import sys
int_args = map( int, sys.argv[1:] )
```



Rückgabewerte



- Die return-Anweisung gibt einen Wert aus der Funktion zurück

```
a = 3
def square( x ):
    square = x*x
    return square

print a
a = square( a )
print a
```

Ausgabe

```
3
9
```

- Mehrere Werte können als Tupel zurückgegeben werden

```
def square_cube( x ):
    square = x*x
    cube = x*x*x
    return ( square, cube )

a = 3
x, y = square( a )      # keine Klammern auf lhs!
```



Scope von Variablennamen



- Variablennamen in Funktionen nur innerhalb der Funktion gültig

```
a = 3
def foo( x ):
    a = 5

print a
foo( a )
print a
```

Ausgabe

```
3
3
```

- Auf Variablen außerhalb greift man per `global`-Anweisung zu

```
a = 3
def foo( x ):
    global a
    a = 5

print a
foo( a )
print a
```

Ausgabe

```
3
5
```



Beispiel: Berechnung der Tages



```
def Day( day, month, year ) :  
    days = ["Mo", "Di", "Mi", "Do", "Fr", "Sa", "So"]  
    y = year - (14 - month) / 12  
    x = y + y/4 - y/100 + y/400  
    m = month + 12 * ((14 - month) / 12) - 2  
    d = (day + x + (31*m)/12) % 7  
    return days[d]  
  
print Day( 24, 12, 2005 )
```

Ausgabe

Sa



Module



- Wenn Programme zu lang werden, kann man sie in mehrere Dateien unterteilen, um sie besser warten zu können
- Python erlaubt es, Definitionen in eine Datei zu setzen und sie als Modul zu benutzen
- Um ein Modul zu erzeugen schreibt man die Def's in einen File, der denselben Namen wie das Modul und Suffix **.py** hat

```
# File: div.py # bildet divmod() nach  
def divide( a, b ) :  
    q = a/b  
    r = a-q*b # Wenn a und b ganzzahlig, dann auch q  
    return (q, r) # liefert ein Tuple
```



Verwendung von Modulen

- Um ein Modul zu verwenden benutzt man die `import`-Anweisung
- Um auf eine Funktion aus einem Module zuzugreifen, stellt man ihr den Modulnamen voran

```
import div
a, b = div.divide( 100, 35 )
```

- Um spezielle, einzelne Definitionen in den aktuellen Namensraum zu importieren benutzt man die `from`-Anweisung

```
from div import divide
a, b = divide( 100, 35 )
```

- den gesamten Inhalt eines Moduls importiert man mit:

```
from div import *
```

kein `div.` mehr nötig



Modul-Suchpfad

- Beim Laden der Module sucht der Interpreter in einer Liste von Verzeichnissen, die in der Liste `sys.path` definiert ist:

```
>>> import sys
>>> sys.path
['', '/usr/local/lib/python1.5/',
'/usr/local/lib/python1.5/test',
'/usr/local/lib/python1.5/plat-sunos5',
'/usr/local/lib/python1.5/lib-tk',
'/usr/local/lib/python1.5/lib-dynload',
'/usr/local/lib/site-python']
```

- Neue Verzeichnisse fügt man dem Suchpfad durch einen Eintrag in die Liste hinzu, z.B.

```
>>> sys.path.append( "." )
```



Modul `math`



- Das Modul `math` definiert mathematische Standardfunktionen für Floating-Point-Zahlen

Einige Funktionen des `math`-Moduls

<code>ceil(x)</code>	Ergibt nächstgrößere ganze Zahl von <code>x</code> .
<code>cos(x)</code>	Ergibt Cosinus von <code>x</code> .
<code>exp(x)</code>	Ergibt e^{**x} .
<code>fabs(x)</code>	Ergibt Betrag von <code>x</code> .
<code>floor(x)</code>	Ergibt nächstkleinere ganze Zahl von <code>x</code> .
<code>fmod(x, y)</code>	Ergibt $x \% y$.
<code>frexp(x)</code>	Ergibt positive Mantisse und Exponenten von <code>x</code> .
<code>hypot(x, y)</code>	Ergibt Euklidischen Abstand, $\sqrt{x^2+y^2}$.
<code>ldexp(x, i)</code>	Ergibt $x * (2^{**i})$.
<code>log(x)</code>	Ergibt natürlichen Logarithmus von <code>x</code> .
<code>log10(x)</code>	Ergibt Logarithmus zur Basis 10 von <code>x</code> .
<code>pow(x, y)</code>	Ergibt x^{**y} .
<code>sin(x)</code>	Ergibt Sinus von <code>x</code> .
<code>sqrt(x)</code>	Ergibt Quadratwurzel von <code>x</code> .
<code>tan(x)</code>	Ergibt Tangens von <code>x</code> .



Modul `cmath`



- Python hat neben Ganzzahlen und FP-Zahlen auch komplexe Zahlen als Typ direkt eingebaut
- Zahlen mit `j` am Ende interpretiert Python als komplexe Zahlen
- Komplexe Zahlen mit Real- und Imaginärteil erzeugt man durch Addition, also z.B. $c = 1.2 + 12.24j$
- Das Modul `cmath` definiert mathematische Standardfunktionen für komplexe Zahlen

Einige Funktionen des `cmath`-Moduls

<code>cos(x)</code>	Ergibt Cosinus von <code>x</code> .
<code>exp(x)</code>	Ergibt e^{**x} .
<code>log(x)</code>	Ergibt natürlichen Logarithmus von <code>x</code> .
<code>log10(x)</code>	Ergibt Logarithmus zur Basis 10 von <code>x</code> .
<code>sin(x)</code>	Ergibt Sinus von <code>x</code> .
<code>sqrt(x)</code>	Ergibt Quadratwurzel von <code>x</code> .
<code>tan(x)</code>	Ergibt Tangens von <code>x</code> .



Beispiel: Mandelbrotmenge



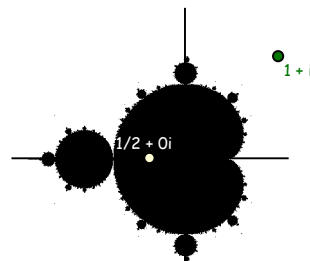
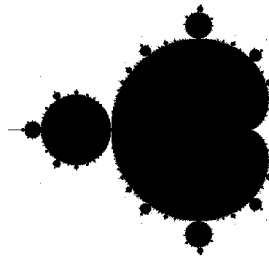
- Menge von Punkten M in der komplexen Ebene:

- Bilde zu jedem $c \in \mathbb{C}$ die (unendliche) Folge

$$z_{i+1} = z_i^2 + c, \quad z_0 = 0$$

- Definiere Mandelbrot-Menge

$$M = \{c \in \mathbb{C} \mid \text{Folge } (z_i) \text{ bleibt beschränkt} \}$$



i	z_i
0	$1 + i$
1	$1 + 3i$
2	$-7 + 7i$
3	$1 - 97i$
4	$-9407 - 193i$
5	$88454401 + 3631103i$

$c = 1 + i$ ist nicht in der Mandelbrotmenge enthalten

i	z_i
0	$-1/2$
1	$-1/4$
2	$-7/16$
3	1
4	$-79/256$
5	$-26527/65536$

$c = -1/2$ ist in der Mandelbrotmenge enthalten



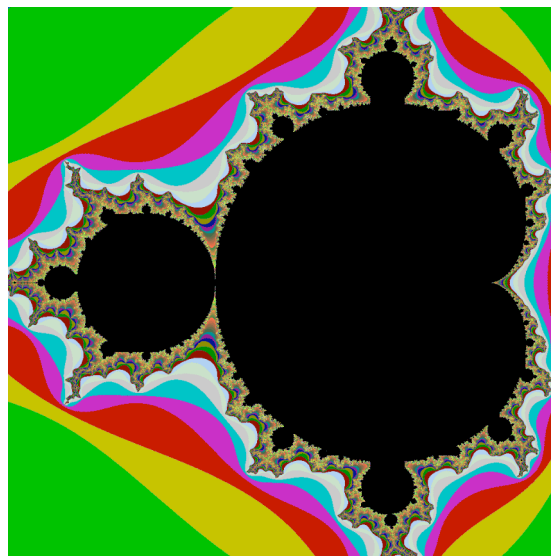
Visualisierung der Mandelbrotmenge



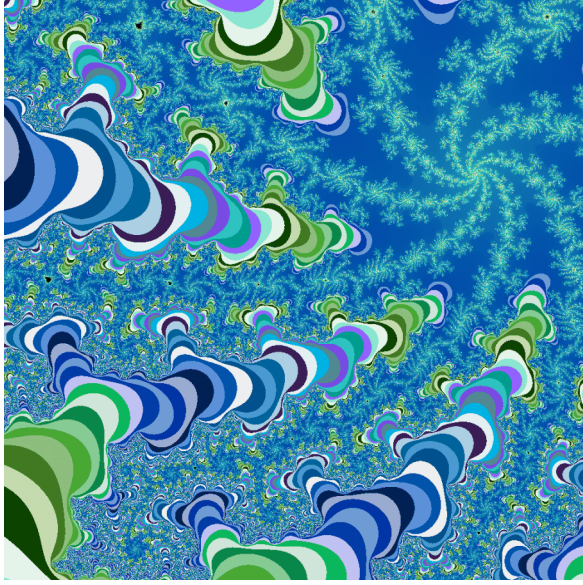
- Färbe Pixel (x, y) schwarz falls $z = x + iy$ in der Menge ist, sonst weiß
- Einschränkungen in der Praxis:
 - Man kann nicht unendlich viele Punkte zeichnen
 - Man kann nicht unendlich oft iterieren
- Deswegen: Approximative Lösung
 - Wähle eine endliche Menge von Punkten aus
 - Iteriere N mal
 - Satz (o. Bew.): Ist $|z_t| > 2$ für ein t , dann ist c nicht in der Mandelbrotmenge
 - Es gilt (fast immer): Ist $|z_{256}| \leq 2$ dann ist c "wahrscheinlich" in der Mandelbrotmenge enthalten
- Schönere Bilder erhält man, wenn man die Punkte zusätzlich färbt:
 - Färbe c abhängig von der Anzahl an Iterationen t die nötig waren, bis $|z_t| > 2$ wurde.



Mandelbrotmengen-Impressionen



Mandelbrotmengen-Impressionen



Prof. Dr. G. Zachmann Informatik 1 - WS 05/06 Einführung in Python, Teil 2 40

Modul `random`

- Das Modul `random` erzeugt Pseudo-Zufallszahlen

Einige Funktionen des <code>random</code> -Moduls	
<code>choice(seq)</code>	Gibt zufälliges Element einer Sequenz <code>seq</code> zurück
<code>random()</code>	Gibt Zufallszahl zwischen 0 und 1 aus
<code>uniform(a, b)</code>	Gibt normalverteilte Zufallszahl aus dem Intervall <code>[a, b)</code> zurück
<code>randint(a, b)</code>	Gibt ganzzahlige Zufallszahl aus dem Intervall <code>[a, b]</code> zurück
<code>seed([x])</code>	Initialisiert den Zufallszahl-Generator. Falls <code>x</code> nicht explizit angegeben wird, wird einfach die aktuelle Systemzeit verwendet

Prof. Dr. G. Zachmann Informatik 1 - WS 05/06 Einführung in Python, Teil 2 41



Python Imaging Library (PIL)

- Bibliothek zum Erzeugen, Konvertieren, Bearbeiten, usw von Bildern
- Die Bibliothek enthält mehrere Module
 - Image Modul
 - ImageDraw Modul
 - ImageFile Modul
 - ImageFilter Modul
 - ImageColor Modul
 - ImageWin Modul
 - ...



Module `Image` der PIL

- Stellt grundlegende Funktionen zur Verfügung

Einige Funktionen des `Image`-Moduls

`new(mode, size)` Erzeugt neues Bild. `mode` ist ein String der das verwendete Pixelformat beschreibt (z.B. "RGB", "CMYK"), `size` ist ein 2-Tupel, durch welches Höhe und Breite des Bildes in Pixeln angegeben werden

`new(mode, size, color)` Wie oben mit einem zusätzlichen 3-Tupel für die Farbtiefe.

`putpixel(xy, color)` Setzt den Pixel an der Position (`x`, `y`) auf den angegebenen Farbwert

`show()` Zeigt das Bild an. Die Ausgabe ist abhängig vom verwendeten Betriebssystem

`save(outfile, options)` Speichert ein Bild in der Datei mit dem Namen `outfile`. Zusätzlich können noch Optionen angegeben werden





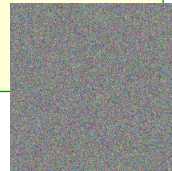
Beispiel

```
# import Libraries
import Image
import random

# Create new Image
im = Image.new("RGB", (512, 512), (256, 256, 256) )

# Set some Pixels randomly in the Image
for i in range( 0, 512 ):
    for j in range( 0, 512 ):
        r = random.randint(0, 256)
        g = random.randint(0, 256)
        b = random.randint(0, 256)
        im.putpixel( (i, j), (r, g, b) )

# Finally: Show the image
im.show()
```



Modul ImageDraw der PIL

- Einfache Funktionen zum Erzeugen von 2D-Grafiken:
 - `ellipse(xy, options)` Erzeugt eine Ellipse
 - `line(xy, options)` Erzeugt eine Linie
 - `point(xy, options)` Erzeugt einen Punkt
 - `polygon(xy, options)` Erzeugt ein Polygon
 - `rectangle(box, options)` Erzeugt ein Rechteck
 - `text(position, string, options)` Erzeugt eine Text



Beispiel

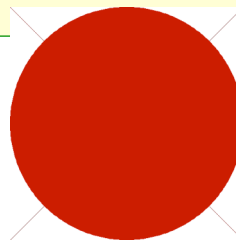


```
import Image, ImageDraw

im = Image.new("RGB", (512, 512), (256, 256, 256) )

draw = ImageDraw.Draw(im)
draw.line((0, 0) + im.size, fill=128)
draw.line((0, im.size[1], im.size[0], 0), fill=100)
draw.ellipse ((0, 0) + im.size, fill=200)
del draw

im.show()
```



Unit-Tests in Python



- Unit-Test
 - Einfachste Form von Software-Test aus dem Software-Engineering
 - Unit = Funktion oder Klasse
 - Testet, ob Ist-Ausgabe der Soll-Ausgabe entspricht
 - Wird normalerweise vom Programmierer der Funktion/Klasse gleich mitgeschrieben
 - er weiß am besten, was rauskommen muß
 - Dient gleichzeitig der gedanklichen Unterstützung beim Aufstellen der Spezifikation der Funktion / Klasse
- Unit-Tests können später automatisiert im Batch ablaufen
 - Stellt sicher, daß Einzelteile der Software noch das tun, was sie sollen
 - Stellt sicher, daß im Code-Repository immer eine korrekte Version ist



Integration von Unit-Tests im Modul selbst (der `__name__`-Trick)



- Jedes Python-Modul besitzt einen eigenen Namen
 - Innerhalb eines Moduls ist der Modulname (als String) als Wert der globalen Variablen `__name__` verfügbar.
 - Die Variable `__main__` enthält den Namen des Hauptprogramms
- Dadurch lassen sich in Python sehr leicht Unit-Tests **direkt im Modul** implementieren:
 - Bestimmte Teile eines Moduls werden nur dann ausgeführt, wenn man es als eigenständiges Programm startet
 - Beim Import in ein anderes Modul werden diese Teile nicht ausgeführt

```
if __name__ == "__main__":  
    print 'This program is being run by itself'  
else:  
    print 'I am being imported from another module'
```



Beispiel



```
def ggt(a,b):  
    while b != 0:  
        a, b = b, a%b  
    return a  
  
def test_ggt():  
    if ggt(100, 0) == 100:  
        print "test1 passed"  
    else:  
        print "test1 failed"  
    if ggt(43, 51) == 1:  
        print "test2 passed"  
    else:  
        print "test2 failed"  
    if ggt(10, 5) == 5:  
        print "test3 passed"  
    else:  
        print "test3 failed"  
  
if __name__ == "__main__":  
    test_ggt()
```