Informatik I: Einführung in die Programmierung

4. Funktionen: Aufrufe und Definitionen

UNI FREIBURG

Albert-Ludwigs-Universität Freiburg

Bernhard Nebel

20./24. Oktober 2017

UNI FREIBURG

Funktionsaufrufe

Funktionsaufrufe

UNI FREIBURG

- Innerhalb der Mathematik sind Funktionen Abbildungen von einem Definitionsbereich in einen Bildbereich.
- Innerhalb von Programmiersprachen ist eine Funktion ein Programmstück (meistens mit einem Namen versehen).
- Normalerweise erwartet eine Funktion Argumente und gibt einen Funktionswert (oder *Rückgabewert*) zurück, und berechnet also eine Abbildung – aber Seiteneffekte Abhängigkeit von globalen Variablen sind möglich.
- `type`-Funktion:

Python-Interpreter

```
>>> type(42)
<class 'int'>
```

- Funktion mit variabler Anzahl von Argumenten und ohne Rückgabewert (aber mit Seiteneffekt): `print`
- Funktion ohne Argumente und ohne Rückgabewert: `exit`

Standardfunktionen: Typen-Konversion

UNI FREIBURG

Mit den Funktionen `int`, `float`, `complex`, `str` kann man „passende“ Werte in den jeweiligen Typ umwandeln.
Umwandlung nach `int` durch „Abschneiden“.

Python-Interpreter

```
>>> int(-2.6)
```

Standardfunktionen: Typen-Konversion

UNI FREIBURG

Mit den Funktionen `int`, `float`, `complex`, `str` kann man „passende“ Werte in den jeweiligen Typ umwandeln.
Umwandlung nach `int` durch „Abschneiden“.

Python-Interpreter

```
>>> int(-2.6)
-2
>>>
```

Standardfunktionen: Typen-Konversion

UNI FREIBURG

Mit den Funktionen `int`, `float`, `complex`, `str` kann man „passende“ Werte in den jeweiligen Typ umwandeln.
Umwandlung nach `int` durch „Abschneiden“.

Python-Interpreter

```
>>> int(-2.6)
-2
>>> int('vier')
```

Standardfunktionen: Typen-Konversion

UNI FREIBURG

Mit den Funktionen `int`, `float`, `complex`, `str` kann man „passende“ Werte in den jeweiligen Typ umwandeln.
Umwandlung nach `int` durch „Abschneiden“.

Python-Interpreter

```
>>> int(-2.6)
-2
>>> int('vier')
 File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() ...
>>>
```

Standardfunktionen: Typen-Konversion

UNI FREIBURG

Mit den Funktionen `int`, `float`, `complex`, `str` kann man „passende“ Werte in den jeweiligen Typ umwandeln.
Umwandlung nach `int` durch „Abschneiden“.

Python-Interpreter

```
>>> int(-2.6)
-2
>>> int('vier')
 File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() ...
>>> complex('42')
```

Standardfunktionen: Typen-Konversion

UNI FREIBURG

Mit den Funktionen `int`, `float`, `complex`, `str` kann man „passende“ Werte in den jeweiligen Typ umwandeln.
Umwandlung nach `int` durch „Abschneiden“.

Python-Interpreter

```
>>> int(-2.6)
-2
>>> int('vier')
 File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() ...
>>> complex('42')
(42+0j)
>>>
```

Standardfunktionen: Typen-Konversion

UNI FREIBURG

Mit den Funktionen `int`, `float`, `complex`, `str` kann man „passende“ Werte in den jeweiligen Typ umwandeln.
Umwandlung nach `int` durch „Abschneiden“.

Python-Interpreter

```
>>> int(-2.6)
-2
>>> int('vier')
 File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() ...
>>> complex('42')
(42+0j)
>>> float(4)
```

Standardfunktionen: Typen-Konversion

UNI FREIBURG

Mit den Funktionen `int`, `float`, `complex`, `str` kann man „passende“ Werte in den jeweiligen Typ umwandeln.
Umwandlung nach `int` durch „Abschneiden“.

Python-Interpreter

```
>>> int(-2.6)
-2
>>> int('vier')
 File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() ...
>>> complex('42')
(42+0j)
>>> float(4)
4.0
>>>
```

Standardfunktionen: Typen-Konversion

UNI FREIBURG

Mit den Funktionen `int`, `float`, `complex`, `str` kann man „passende“ Werte in den jeweiligen Typ umwandeln.
Umwandlung nach `int` durch „Abschneiden“.

Python-Interpreter

```
>>> int(-2.6)
-2
>>> int('vier')
 File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() ...
>>> complex('42')
(42+0j)
>>> float(4)
4.0
>>> str(42)
```

Standardfunktionen: Typen-Konversion

UNI FREIBURG

Mit den Funktionen `int`, `float`, `complex`, `str` kann man „passende“ Werte in den jeweiligen Typ umwandeln.
Umwandlung nach `int` durch „Abschneiden“.

Python-Interpreter

```
>>> int(-2.6)
-2
>>> int('vier')
 File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() ...
>>> complex('42')
(42+0j)
>>> float(4)
4.0
>>> str(42)
'42'
```

Standardfunktionen: Numerische Funktionen

UNI FREIBURG

`abs` liefert den Absolutwert (auch bei `complex`), `round` rundet, und `pow` berechnet die Exponentation bei zwei Argumenten oder die Exponentation modulo dem dritten Argument.

Python-Interpreter

```
>>> abs(-2)
```

Standardfunktionen: Numerische Funktionen

UNI FREIBURG

`abs` liefert den Absolutwert (auch bei `complex`), `round` rundet, und `pow` berechnet die Exponentation bei zwei Argumenten oder die Exponentation modulo dem dritten Argument.

Python-Interpreter

```
>>> abs(-2)
2
>>>
```

Standardfunktionen: Numerische Funktionen

UNI FREIBURG

`abs` liefert den Absolutwert (auch bei `complex`), `round` rundet, und `pow` berechnet die Exponentation bei zwei Argumenten oder die Exponentation modulo dem dritten Argument.

Python-Interpreter

```
>>> abs(-2)
2
>>> abs(1+1j)
```

Standardfunktionen: Numerische Funktionen

UNI FREIBURG

`abs` liefert den Absolutwert (auch bei `complex`), `round` rundet, und `pow` berechnet die Exponentation bei zwei Argumenten oder die Exponentation modulo dem dritten Argument.

Python-Interpreter

```
>>> abs(-2)
2
>>> abs(1+1j)
1.4142135623730951
>>>
```

Standardfunktionen: Numerische Funktionen

UNI FREIBURG

`abs` liefert den Absolutwert (auch bei `complex`), `round` rundet, und `pow` berechnet die Exponentation bei zwei Argumenten oder die Exponentation modulo dem dritten Argument.

Python-Interpreter

```
>>> abs(-2)
2
>>> abs(1+1j)
1.4142135623730951
>>> round(2.500001)
```

Standardfunktionen: Numerische Funktionen

UNI FREIBURG

`abs` liefert den Absolutwert (auch bei `complex`), `round` rundet, und `pow` berechnet die Exponentation bei zwei Argumenten oder die Exponentation modulo dem dritten Argument.

Python-Interpreter

```
>>> abs(-2)
2
>>> abs(1+1j)
1.4142135623730951
>>> round(2.500001)
3
>>>
```

Standardfunktionen: Numerische Funktionen

UNI FREIBURG

`abs` liefert den Absolutwert (auch bei `complex`), `round` rundet, und `pow` berechnet die Exponentation bei zwei Argumenten oder die Exponentation modulo dem dritten Argument.

Python-Interpreter

```
>>> abs(-2)
2
>>> abs(1+1j)
1.4142135623730951
>>> round(2.500001)
3
>>> pow(2, 3)
```

Standardfunktionen: Numerische Funktionen

UNI FREIBURG

`abs` liefert den Absolutwert (auch bei `complex`), `round` rundet, und `pow` berechnet die Exponentation bei zwei Argumenten oder die Exponentation modulo dem dritten Argument.

Python-Interpreter

```
>>> abs(-2)
2
>>> abs(1+1j)
1.4142135623730951
>>> round(2.500001)
3
>>> pow(2, 3)
8
>>>
```

Standardfunktionen: Numerische Funktionen

UNI FREIBURG

`abs` liefert den Absolutwert (auch bei `complex`), `round` rundet, und `pow` berechnet die Exponentation bei zwei Argumenten oder die Exponentation modulo dem dritten Argument.

Python-Interpreter

```
>>> abs(-2)
2
>>> abs(1+1j)
1.4142135623730951
>>> round(2.500001)
3
>>> pow(2, 3)
8
>>> pow(2, 3, 4)
```

Standardfunktionen: Numerische Funktionen

UNI FREIBURG

`abs` liefert den Absolutwert (auch bei `complex`), `round` rundet, und `pow` berechnet die Exponentation bei zwei Argumenten oder die Exponentation modulo dem dritten Argument.

Python-Interpreter

```
>>> abs(-2)
2
>>> abs(1+1j)
1.4142135623730951
>>> round(2.500001)
3
>>> pow(2, 3)
8
>>> pow(2, 3, 4)
0
```

Standardfunktionen: Zeichen-Konversion

UNI FREIBURG

Mit den Funktionen `chr` und `ord` kann man Zahlen in Unicode-Zeichen und umgekehrt umwandeln, wobei in Python Zeichen identisch mit einbuchstabigen Strings sind:

Python-Interpreter

```
>>> chr(42)
```

Standardfunktionen: Zeichen-Konversion

UNI FREIBURG

Mit den Funktionen `chr` und `ord` kann man Zahlen in Unicode-Zeichen und umgekehrt umwandeln, wobei in Python Zeichen identisch mit einbuchstabigen Strings sind:

Python-Interpreter

```
>>> chr(42)
'*'
>>>
```

Standardfunktionen: Zeichen-Konversion

UNI FREIBURG

Mit den Funktionen `chr` und `ord` kann man Zahlen in Unicode-Zeichen und umgekehrt umwandeln, wobei in Python Zeichen identisch mit einbuchstabigen Strings sind:

Python-Interpreter

```
>>> chr(42)
'*'
>>> chr(255)
```

Standardfunktionen: Zeichen-Konversion

UNI FREIBURG

Mit den Funktionen `chr` und `ord` kann man Zahlen in Unicode-Zeichen und umgekehrt umwandeln, wobei in Python Zeichen identisch mit einbuchstabigen Strings sind:

Python-Interpreter

```
>>> chr(42)
'*'
>>> chr(255)
'ÿ'
>>>
```

Standardfunktionen: Zeichen-Konversion

UNI FREIBURG

Mit den Funktionen `chr` und `ord` kann man Zahlen in Unicode-Zeichen und umgekehrt umwandeln, wobei in Python Zeichen identisch mit einbuchstabigen Strings sind:

Python-Interpreter

```
>>> chr(42)
'*'
>>> chr(255)
'ÿ'
>>> ord('*')
```

Standardfunktionen: Zeichen-Konversion

UNI FREIBURG

Mit den Funktionen `chr` und `ord` kann man Zahlen in Unicode-Zeichen und umgekehrt umwandeln, wobei in Python Zeichen identisch mit einbuchstabigen Strings sind:

Python-Interpreter

```
>>> chr(42)
'*'
>>> chr(255)
'ÿ'
>>> ord('*')
42
>>>
```

Standardfunktionen: Zeichen-Konversion

UNI FREIBURG

Mit den Funktionen `chr` und `ord` kann man Zahlen in Unicode-Zeichen und umgekehrt umwandeln, wobei in Python Zeichen identisch mit einbuchstabigen Strings sind:

Python-Interpreter

```
>>> chr(42)
'*'
>>> chr(255)
'ÿ'
>>> ord('*')
42
>>> ord('**')
```

Standardfunktionen: Zeichen-Konversion

UNI FREIBURG

Mit den Funktionen `chr` und `ord` kann man Zahlen in Unicode-Zeichen und umgekehrt umwandeln, wobei in Python Zeichen identisch mit einbuchstabigen Strings sind:

Python-Interpreter

```
>>> chr(42)
'*'
>>> chr(255)
'ÿ'
>>> ord('*')
42
>>> ord('**')
Traceback (most recent call last): ...
TypeError: ord() expected a character, but string of
length 2 found
```

Kleiner Exkurs: Zeichenkodierung

UNI FREIBURG

- Computer kann man dafür nutzen, Berechnungen durchzuführen.

Kleiner Exkurs: Zeichenkodierung

UNI FREIBURG

- Computer kann man dafür nutzen, Berechnungen durchzuführen.
- Sehr früh hat man aber auch begonnen, mit dem Computer Texte zu verarbeiten

Kleiner Exkurs: Zeichenkodierung

UNI FREIBURG

- Computer kann man dafür nutzen, Berechnungen durchzuführen.
- Sehr früh hat man aber auch begonnen, mit dem Computer Texte zu verarbeiten
- Wie stellt man Texte im Computer dar?

Kleiner Exkurs: Zeichenkodierung

UNI FREIBURG

- Computer kann man dafür nutzen, Berechnungen durchzuführen.
- Sehr früh hat man aber auch begonnen, mit dem Computer Texte zu verarbeiten
- Wie stellt man Texte im Computer dar?
- Man weist jedem Buchstaben einen Zahlenwert zu. Texte sind dann Sequenzen von solchen Codezahlen.

Kleiner Exkurs: Zeichenkodierung

UNI FREIBURG

- Computer kann man dafür nutzen, Berechnungen durchzuführen.
- Sehr früh hat man aber auch begonnen, mit dem Computer Texte zu verarbeiten
- Wie stellt man Texte im Computer dar?
- Man weist jedem Buchstaben einen Zahlenwert zu. Texte sind dann Sequenzen von solchen Codezahlen.
- Damit wird dann auch **Textverabeitung** zu einer Berechnung.

ASCII

- Einer der ersten Zeichenkodes war ASCII (American Standard Code for Information Interchange) – entwickelt für Fernschreiber.

USASCII code chart

Bits b_7 b_6 b_5 →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4 b_3 b_2 b_1 ↓ Row / Column →	0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	FS	,	<	L	\	l	|
1 1 0 1	13	CR	GS	-	=	M	]	m	}
1 1 1 0	14	SO	RS	.	>	N	^	n	~
1 1 1 1	15	SI	US	/	?	O	_	o	DEL

ASCII

- Einer der ersten Zeichenkodes war ASCII (American Standard Code for Information Interchange) – entwickelt für Fernschreiber.

USASCII code chart

Bits b_7 b_6 b_5 →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4 b_3 b_2 b_1 ↓ Row / Column →	0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	FS	,	<	L	\	l	|
1 1 0 1	13	CR	GS	-	=	M	]	m	}
1 1 1 0	14	SO	RS	.	>	N	^	n	~
1 1 1 1	15	SI	US	/	?	O	_	o	DEL

- Benötigt 7 Bits und enthält alle druckbaren Zeichen der englischen Sprache sowie nicht-druckbare Steuerzeichen (z.B. Zeilenwechsel).

Erweitertes ASCII

UNI FREIBURG

- In anderen Sprachen wurden zusätzliche Zeichen benötigt.

Erweitertes ASCII

UNI FREIBURG

- In anderen Sprachen wurden zusätzliche Zeichen benötigt.
- Da mittlerweile praktisch alle Rechner 8-Bit-Bytes als kleinste Speichereinheit nutzen, kann man die höherwertigen Kodes (128–255) für Erweiterungen nutzen.

Erweitertes ASCII

UNI FREIBURG

- In anderen Sprachen wurden zusätzliche Zeichen benötigt.
- Da mittlerweile praktisch alle Rechner 8-Bit-Bytes als kleinste Speichereinheit nutzen, kann man die höherwertigen Kodes (128–255) für Erweiterungen nutzen.
- Diverse Erweiterungen, z.B. ISO-Latin-1 (mit Umlauten) usw.

Erweitertes ASCII

UNI FREIBURG

- In anderen Sprachen wurden zusätzliche Zeichen benötigt.
- Da mittlerweile praktisch alle Rechner 8-Bit-Bytes als kleinste Speichereinheit nutzen, kann man die höherwertigen Kodes (128–255) für Erweiterungen nutzen.
- Diverse Erweiterungen, z.B. ISO-Latin-1 (mit Umlauten) usw.
- Auf dem IBM-PC gab es andere Erweiterungen.

Erweitertes ASCII

UNI FREIBURG

- In anderen Sprachen wurden zusätzliche Zeichen benötigt.
- Da mittlerweile praktisch alle Rechner 8-Bit-Bytes als kleinste Speichereinheit nutzen, kann man die höherwertigen Kodes (128–255) für Erweiterungen nutzen.
- Diverse Erweiterungen, z.B. ISO-Latin-1 (mit Umlauten) usw.
- Auf dem IBM-PC gab es andere Erweiterungen.
- Sprachen, die nicht auf dem lateinischen Alphabet basieren, haben große Probleme.

Unicode

UNI FREIBURG

- Um für alle Sprachräume einen einheitlichen Zeichencode zu haben, wurde Unicode entwickelt (Version 1.0 im Jahr 1991).

Unicode

UNI FREIBURG

- Um für alle Sprachräume einen einheitlichen Zeichencode zu haben, wurde Unicode entwickelt (Version 1.0 im Jahr 1991).
- Mittlerweile (Juni 2015, Version 8.0) enthält Unicode 120737 Codepoints.

Unicode

UNI FREIBURG

- Um für alle Sprachräume einen einheitlichen Zeichencode zu haben, wurde Unicode entwickelt (Version 1.0 im Jahr 1991).
- Mittlerweile (Juni 2015, Version 8.0) enthält Unicode 120737 Codepoints.
- Organisiert in 17 Ebenen mit jeweils 2^{16} Codepoints (manche allerdings ungenutzt)

Unicode

UNI FREIBURG

- Um für alle Sprachräume einen einheitlichen Zeichencode zu haben, wurde Unicode entwickelt (Version 1.0 im Jahr 1991).
- Mittlerweile (Juni 2015, Version 8.0) enthält Unicode 120737 Codepoints.
- Organisiert in 17 Ebenen mit jeweils 2^{16} Codepoints (manche allerdings ungenutzt)
- Die ersten 128 Codepoints stimmen mit ASCII überein, die ersten 256 mit ISO-Latin-1.

UTF-32, UTF-16 und UTF-8

UNI FREIBURG

- Man kann Unicode-Zeichen als eine 32-Bit-Zahl darstellen (UTF-32 oder UCS-4).

UTF-32, UTF-16 und UTF-8

UNI FREIBURG

- Man kann Unicode-Zeichen als eine 32-Bit-Zahl darstellen (UTF-32 oder UCS-4).
- Da man meist nur die Ebene 0 benötigt, ist es effizienter, die Kodierung UTF-16 einzusetzen, bei der die Ebene 0 direkt als 16-Bit-Zahl kodiert wird. Zeichen aus anderen Ebenen benötigen 32 Bit.

UTF-32, UTF-16 und UTF-8

UNI FREIBURG

- Man kann Unicode-Zeichen als eine 32-Bit-Zahl darstellen (UTF-32 oder UCS-4).
- Da man meist nur die Ebene 0 benötigt, ist es effizienter, die Kodierung UTF-16 einzusetzen, bei der die Ebene 0 direkt als 16-Bit-Zahl kodiert wird. Zeichen aus anderen Ebenen benötigen 32 Bit.
- Im WWW wird meist UTF-8 eingesetzt:

Unicode	UTF-8 binär
0–127	0xxxxxxx
128–2047	110xxxxx 10xxxxxx
2048–65535	1110xxxx 10xxxxxx 10xxxxxx
65536–1114111	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

UTF-32, UTF-16 und UTF-8

UNI FREIBURG

- Man kann Unicode-Zeichen als eine 32-Bit-Zahl darstellen (UTF-32 oder UCS-4).
- Da man meist nur die Ebene 0 benötigt, ist es effizienter, die Kodierung UTF-16 einzusetzen, bei der die Ebene 0 direkt als 16-Bit-Zahl kodiert wird. Zeichen aus anderen Ebenen benötigen 32 Bit.
- Im WWW wird meist UTF-8 eingesetzt:

Unicode	UTF-8 binär
0–127	0xxxxxxx
128–2047	110xxxxx 10xxxxxx
2048–65535	1110xxxx 10xxxxxx 10xxxxxx
65536–1114111	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

- Wie kommen die komischen Zeichen auf Webseiten zustande?

UTF-32, UTF-16 und UTF-8

UNI FREIBURG

- Man kann Unicode-Zeichen als eine 32-Bit-Zahl darstellen (UTF-32 oder UCS-4).
- Da man meist nur die Ebene 0 benötigt, ist es effizienter, die Kodierung UTF-16 einzusetzen, bei der die Ebene 0 direkt als 16-Bit-Zahl kodiert wird. Zeichen aus anderen Ebenen benötigen 32 Bit.
- Im WWW wird meist UTF-8 eingesetzt:

Unicode	UTF-8 binär
0–127	0xxxxxxx
128–2047	110xxxxx 10xxxxxx
2048–65535	1110xxxx 10xxxxxx 10xxxxxx
65536–1114111	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

- Wie kommen die komischen Zeichen auf Webseiten zustande?
- Oft sind ISO-Latin-1/UTF-8 Verwechslungen der Grund!

UNI FREIBURG

Mathematische Funktionen

Mathematische Funktionen: Das Math-Modul

UNI FREIBURG

- Natürlich wollen wir Funktionen wie `sin` verwenden. Die muss man in Python aber erst durch Importieren des Mathematik-Moduls bekannt machen.
- Danach können wir die Teile des Moduls durch Voranstellen von `math.` nutzen (Punktschreibweise):

Python-Interpreter

```
>>> import math
>>> math.pi
```

Mathematische Funktionen: Das Math-Modul

UNI FREIBURG

- Natürlich wollen wir Funktionen wie `sin` verwenden. Die muss man in Python aber erst durch Importieren des Mathematik-Moduls bekannt machen.
- Danach können wir die Teile des Moduls durch Voranstellen von `math.` nutzen (Punktschreibweise):

Python-Interpreter

```
>>> import math
>>> math.pi
3.141592653589793
>>>
```

Mathematische Funktionen: Das Math-Modul

UNI FREIBURG

- Natürlich wollen wir Funktionen wie `sin` verwenden. Die muss man in Python aber erst durch Importieren des Mathematik-Moduls bekannt machen.
- Danach können wir die Teile des Moduls durch Voranstellen von `math.` nutzen (Punktschreibweise):

Python-Interpreter

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sin(1/4*math.pi)
```

Mathematische Funktionen: Das Math-Modul

UNI FREIBURG

- Natürlich wollen wir Funktionen wie `sin` verwenden. Die muss man in Python aber erst durch Importieren des Mathematik-Moduls bekannt machen.
- Danach können wir die Teile des Moduls durch Voranstellen von `math.` nutzen (Punktschreibweise):

Python-Interpreter

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sin(1/4*math.pi)
0.7071067811865475
>>>
```

Mathematische Funktionen: Das Math-Modul

UNI FREIBURG

- Natürlich wollen wir Funktionen wie `sin` verwenden. Die muss man in Python aber erst durch Importieren des Mathematik-Moduls bekannt machen.
- Danach können wir die Teile des Moduls durch Voranstellen von `math.` nutzen (Punktschreibweise):

Python-Interpreter

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sin(1/4*math.pi)
0.7071067811865475
>>> math.sin(math.pi)
```

Mathematische Funktionen: Das Math-Modul

UNI FREIBURG

- Natürlich wollen wir Funktionen wie `sin` verwenden. Die muss man in Python aber erst durch Importieren des Mathematik-Moduls bekannt machen.
- Danach können wir die Teile des Moduls durch Voranstellen von `math.` nutzen (Punktschreibweise):

Python-Interpreter

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sin(1/4*math.pi)
0.7071067811865475
>>> math.sin(math.pi)
1.2246467991473532e-16
>>>
```

Mathematische Funktionen: Das Math-Modul

UNI FREIBURG

- Natürlich wollen wir Funktionen wie `sin` verwenden. Die muss man in Python aber erst durch Importieren des Mathematik-Moduls bekannt machen.
- Danach können wir die Teile des Moduls durch Voranstellen von `math.` nutzen (Punktschreibweise):

Python-Interpreter

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sin(1/4*math.pi)
0.7071067811865475
>>> math.sin(math.pi)
1.2246467991473532e-16
>>> math.exp(math.log(2))
```

Mathematische Funktionen: Das Math-Modul

UNI FREIBURG

- Natürlich wollen wir Funktionen wie `sin` verwenden. Die muss man in Python aber erst durch Importieren des Mathematik-Moduls bekannt machen.
- Danach können wir die Teile des Moduls durch Voranstellen von `math.` nutzen (Punktschreibweise):

Python-Interpreter

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sin(1/4*math.pi)
0.7071067811865475
>>> math.sin(math.pi)
1.2246467991473532e-16
>>> math.exp(math.log(2))
2.0
```

Mathematische Funktionen: Direkt importieren

UNI FREIBURG

- Die Punktschreibweise verhindert Namenskollisionen, ist aber umständlich
- Mit `from` *module* `import` *name* kann ein Name direkt importiert werden.
- `from` *module* `import *` werden alle Namen direkt importiert.

Python-Interpreter

```
>>> from math import pi
>>> pi
```

Mathematische Funktionen: Direkt importieren

UNI FREIBURG

- Die Punktschreibweise verhindert Namenskollisionen, ist aber umständlich
- Mit `from` *module* `import` *name* kann ein Name direkt importiert werden.
- `from` *module* `import *` werden alle Namen direkt importiert.

Python-Interpreter

```
>>> from math import pi
>>> pi
3.141592653589793
>>>
```

Mathematische Funktionen: Direkt importieren

UNI FREIBURG

- Die Punktschreibweise verhindert Namenskollisionen, ist aber umständlich
- Mit `from` *module* `import` *name* kann ein Name direkt importiert werden.
- `from` *module* `import *` werden alle Namen direkt importiert.

Python-Interpreter

```
>>> from math import pi
>>> pi
3.141592653589793
>>> from math import *
>>> cos(pi)
```

Mathematische Funktionen: Direkt importieren

UNI FREIBURG

- Die Punktschreibweise verhindert Namenskollisionen, ist aber umständlich
- Mit `from` *module* `import` *name* kann ein Name direkt importiert werden.
- `from` *module* `import *` werden alle Namen direkt importiert.

Python-Interpreter

```
>>> from math import pi
>>> pi
3.141592653589793
>>> from math import *
>>> cos(pi)
-1.0
```

UNI FREIBURG

Funktionsdefinitionen

Neue Funktionen definieren

UNI FREIBURG

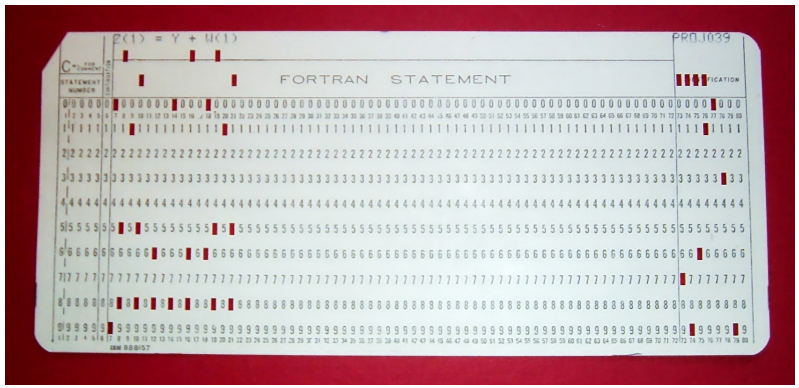
- Mit dem Schlüsselwort `def` kann man eine neue Funktion einführen.
- Nach `def` kommt der Funktionsname gefolgt von der Parameterliste und dann ein Doppelpunkt.
- Nach dem Funktionskopf gibt der Python-Interpreter das Funktionsprompt-Zeichen `...` aus.
- Dann folgt der Funktionsrumpf: *Gleich weit eingerückte Anweisungen*, z.B. Zuweisungen oder Funktionsaufrufe:

Python-Interpreter

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay")
...     print("I sleep all night and I work all day")
...
>>>
```

Einrückungen in Python

- Im Gegensatz zu fast allen anderen Programmiersprachen (außer z.B. FORTRAN, Miranda, Haskell), sind Einrückungen am Zeilenanfang bedeutungstragend.



- In Python ist gleiche Einrückung = zusammen gehöriger Block von Anweisungen
- In den meisten anderen Programmiersprachen durch Klammerung { } oder klammernde Schlüsselwörter.
- Wie viele Leerzeichen sollte man machen?

→ PEP8: 4 Leerzeichen pro Ebene (keine Tabs nutzen!)

Selbst definierte Funktionen nutzen

UNI FREIBURG

- Funktionsnamen müssen den gleichen Regeln folgen wie Variablennamen.
- Tatsächlich verhalten sich Funktionsnamen wie Variablennamen und haben einen entsprechenden Typ.
- Man kann eigene Funktionen wie Standardfunktionen aufrufen

Python-Interpreter

```
>>> print(print_lyrics)
```

Selbst definierte Funktionen nutzen

UNI FREIBURG

- Funktionsnamen müssen den gleichen Regeln folgen wie Variablennamen.
- Tatsächlich verhalten sich Funktionsnamen wie Variablennamen und haben einen entsprechenden Typ.
- Man kann eigene Funktionen wie Standardfunktionen aufrufen

Python-Interpreter

```
>>> print(print_lyrics)
<function print_lyrics at 0x100520560>
```

Selbst definierte Funktionen nutzen

UNI FREIBURG

- Funktionsnamen müssen den gleichen Regeln folgen wie Variablennamen.
- Tatsächlich verhalten sich Funktionsnamen wie Variablennamen und haben einen entsprechenden Typ.
- Man kann eigene Funktionen wie Standardfunktionen aufrufen

Python-Interpreter

```
>>> print(print_lyrics)
<function print_lyrics at 0x100520560>
>>>
```

Selbst definierte Funktionen nutzen

UNI FREIBURG

- Funktionsnamen müssen den gleichen Regeln folgen wie Variablennamen.
- Tatsächlich verhalten sich Funktionsnamen wie Variablennamen und haben einen entsprechenden Typ.
- Man kann eigene Funktionen wie Standardfunktionen aufrufen

Python-Interpreter

```
>>> print(print_lyrics)
<function print_lyrics at 0x100520560>
>>> type(print_lyrics)
```

Selbst definierte Funktionen nutzen

UNI FREIBURG

- Funktionsnamen müssen den gleichen Regeln folgen wie Variablennamen.
- Tatsächlich verhalten sich Funktionsnamen wie Variablennamen und haben einen entsprechenden Typ.
- Man kann eigene Funktionen wie Standardfunktionen aufrufen

Python-Interpreter

```
>>> print(print_lyrics)
<function print_lyrics at 0x100520560>
>>> type(print_lyrics)
<class 'function'>
```

Selbst definierte Funktionen nutzen

UNI FREIBURG

- Funktionsnamen müssen den gleichen Regeln folgen wie Variablennamen.
- Tatsächlich verhalten sich Funktionsnamen wie Variablennamen und haben einen entsprechenden Typ.
- Man kann eigene Funktionen wie Standardfunktionen aufrufen

Python-Interpreter

```
>>> print(print_lyrics)
<function print_lyrics at 0x100520560>
>>> type(print_lyrics)
<class 'function'>
>>>
```

Selbst definierte Funktionen nutzen

UNI FREIBURG

- Funktionsnamen müssen den gleichen Regeln folgen wie Variablennamen.
- Tatsächlich verhalten sich Funktionsnamen wie Variablennamen und haben einen entsprechenden Typ.
- Man kann eigene Funktionen wie Standardfunktionen aufrufen

Python-Interpreter

```
>>> print(print_lyrics)
<function print_lyrics at 0x100520560>
>>> type(print_lyrics)
<class 'function'>
>>> print_lyrics()
```

Selbst definierte Funktionen nutzen

UNI FREIBURG

- Funktionsnamen müssen den gleichen Regeln folgen wie Variablennamen.
- Tatsächlich verhalten sich Funktionsnamen wie Variablennamen und haben einen entsprechenden Typ.
- Man kann eigene Funktionen wie Standardfunktionen aufrufen

Python-Interpreter

```
>>> print(print_lyrics)
<function print_lyrics at 0x100520560>
>>> type(print_lyrics)
<class 'function'>
>>> print_lyrics()
I'm a lumberjack, and I'm okay
I sleep all night and I work all day
```

Selbst definierte Funktionen nutzen

UNI FREIBURG

- Funktionsnamen müssen den gleichen Regeln folgen wie Variablennamen.
- Tatsächlich verhalten sich Funktionsnamen wie Variablennamen und haben einen entsprechenden Typ.
- Man kann eigene Funktionen wie Standardfunktionen aufrufen

Python-Interpreter

```
>>> print(print_lyrics)
<function print_lyrics at 0x100520560>
>>> type(print_lyrics)
<class 'function'>
>>> print_lyrics()
I'm a lumberjack, and I'm okay
I sleep all night and I work all day
>>>
```

Selbst definierte Funktionen nutzen

UNI FREIBURG

- Funktionsnamen müssen den gleichen Regeln folgen wie Variablennamen.
- Tatsächlich verhalten sich Funktionsnamen wie Variablennamen und haben einen entsprechenden Typ.
- Man kann eigene Funktionen wie Standardfunktionen aufrufen

Python-Interpreter

```
>>> print(print_lyrics)
<function print_lyrics at 0x100520560>
>>> type(print_lyrics)
<class 'function'>
>>> print_lyrics()
I'm a lumberjack, and I'm okay
I sleep all night and I work all day
>>> print_lyrics = 42
```

Definierte Funktionen in Funktionsdefinitionen

UNI FREIBURG

Was passiert hier?

Python-Interpreter

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay")
...     print("I sleep all night and I work all day")
...
>>>
```

Definierte Funktionen in Funktionsdefinitionen

UNI FREIBURG

Was passiert hier?

Python-Interpreter

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay")
...     print("I sleep all night and I work all day")
...
>>>
>>> def repeat_lyrics():
...     print_lyrics()
...     print_lyrics()
...
>>>
```

Definierte Funktionen in Funktionsdefinitionen

UNI FREIBURG

Was passiert hier?

Python-Interpreter

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay")
...     print("I sleep all night and I work all day")
...
>>>
>>> def repeat_lyrics():
...     print_lyrics()
...     print_lyrics()
...
>>> repeat_lyrics()
I'm a lumberjack ...
```

Was wird hier exakt ausgeführt?

Argumente und Parameter

UNI FREIBURG

- Auch definierte Funktionen benötigen oft *Argumente*.
- Bei der Definition gibt man *Parameter* an, die beim Aufruf durch die *Argumente* ersetzt werden.

Python-Interpreter

```
>>> michael = 'baldwin'
>>> def print_twice(bruce):
...     print(bruce)
...     print(bruce)
...
>>>
```

Argumente und Parameter

UNI FREIBURG

- Auch definierte Funktionen benötigen oft *Argumente*.
- Bei der Definition gibt man *Parameter* an, die beim Aufruf durch die *Argumente* ersetzt werden.

Python-Interpreter

```
>>> michael = 'baldwin'
>>> def print_twice(bruce):
...     print(bruce)
...     print(bruce)
...
>>> print_twice(michael)
```

Argumente und Parameter

UNI FREIBURG

- Auch definierte Funktionen benötigen oft *Argumente*.
- Bei der Definition gibt man *Parameter* an, die beim Aufruf durch die *Argumente* ersetzt werden.

Python-Interpreter

```
>>> michael = 'baldwin'
>>> def print_twice(bruce):
...     print(bruce)
...     print(bruce)
...
>>> print_twice(michael)
baldwin
baldwin
>>>
```

Argumente und Parameter

UNI FREIBURG

- Auch definierte Funktionen benötigen oft *Argumente*.
- Bei der Definition gibt man *Parameter* an, die beim Aufruf durch die *Argumente* ersetzt werden.

Python-Interpreter

```
>>> michael = 'baldwin'
>>> def print_twice(bruce):
...     print(bruce)
...     print(bruce)
...
>>> print_twice(michael)
baldwin
baldwin
>>> print_twice('Spam ' * 3)
```

Argumente und Parameter

UNI FREIBURG

- Auch definierte Funktionen benötigen oft *Argumente*.
- Bei der Definition gibt man *Parameter* an, die beim Aufruf durch die *Argumente* ersetzt werden.

Python-Interpreter

```
>>> michael = 'baldwin'
>>> def print_twice(bruce):
...     print(bruce)
...     print(bruce)
...
>>> print_twice(michael)
baldwin
baldwin
>>> print_twice('Spam ' * 3)
Spam Spam Spam
Spam Spam Spam
```

Funktionen als Argumente

UNI FREIBURG

- Wir können Funktionen wie andere Werte als Argumente übergeben.

Python-Interpreter

```
>>> def do_twice(f):
...     f()
...     f()
...
>>>
```

Funktionen als Argumente

UNI FREIBURG

- Wir können Funktionen wie andere Werte als Argumente übergeben.

Python-Interpreter

```
>>> def do_twice(f):
...     f()
...     f()
...
>>> do_twice(print_lyrics)
```

Funktionen als Argumente

UNI FREIBURG

- Wir können Funktionen wie andere Werte als Argumente übergeben.

Python-Interpreter

```
>>> def do_twice(f):
...     f()
...     f()
...
>>> do_twice(print_lyrics)
I'm a lumberjack, and I'm okay
I sleep all night and I work all day
I'm a lumberjack, and I'm okay
I sleep all night and I work all day
```

- Das schauen wir uns in der 2. Hälfte des Semesters noch genauer an!

UNI FREIBURG

Namensraum

Namensraum von lokalen Variablen und Parametern

UNI FREIBURG

- Parameter sind nur innerhalb der Funktion sichtbar.
- Lokal (durch Zuweisung) eingeführte Variablen ebenfalls.

Python-Interpreter

```
>>> def cat_twice(part1, part2):
...     cat = part1 + part2
...     print_twice(cat)
...
>>>
```

Namensraum von lokalen Variablen und Parametern

UNI FREIBURG

- Parameter sind nur innerhalb der Funktion sichtbar.
- Lokal (durch Zuweisung) eingeführte Variablen ebenfalls.

Python-Interpreter

```
>>> def cat_twice(part1, part2):
...     cat = part1 + part2
...     print_twice(cat)
...
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
```

Namensraum von lokalen Variablen und Parametern

UNI FREIBURG

- Parameter sind nur innerhalb der Funktion sichtbar.
- Lokal (durch Zuweisung) eingeführte Variablen ebenfalls.

Python-Interpreter

```
>>> def cat_twice(part1, part2):
...     cat = part1 + part2
...     print_twice(cat)
...
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
>>>
```

Namensraum von lokalen Variablen und Parametern

UNI FREIBURG

- Parameter sind nur innerhalb der Funktion sichtbar.
- Lokal (durch Zuweisung) eingeführte Variablen ebenfalls.

Python-Interpreter

```
>>> def cat_twice(part1, part2):
...     cat = part1 + part2
...     print_twice(cat)
...
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
>>> cat
```

Namensraum von lokalen Variablen und Parametern

UNI FREIBURG

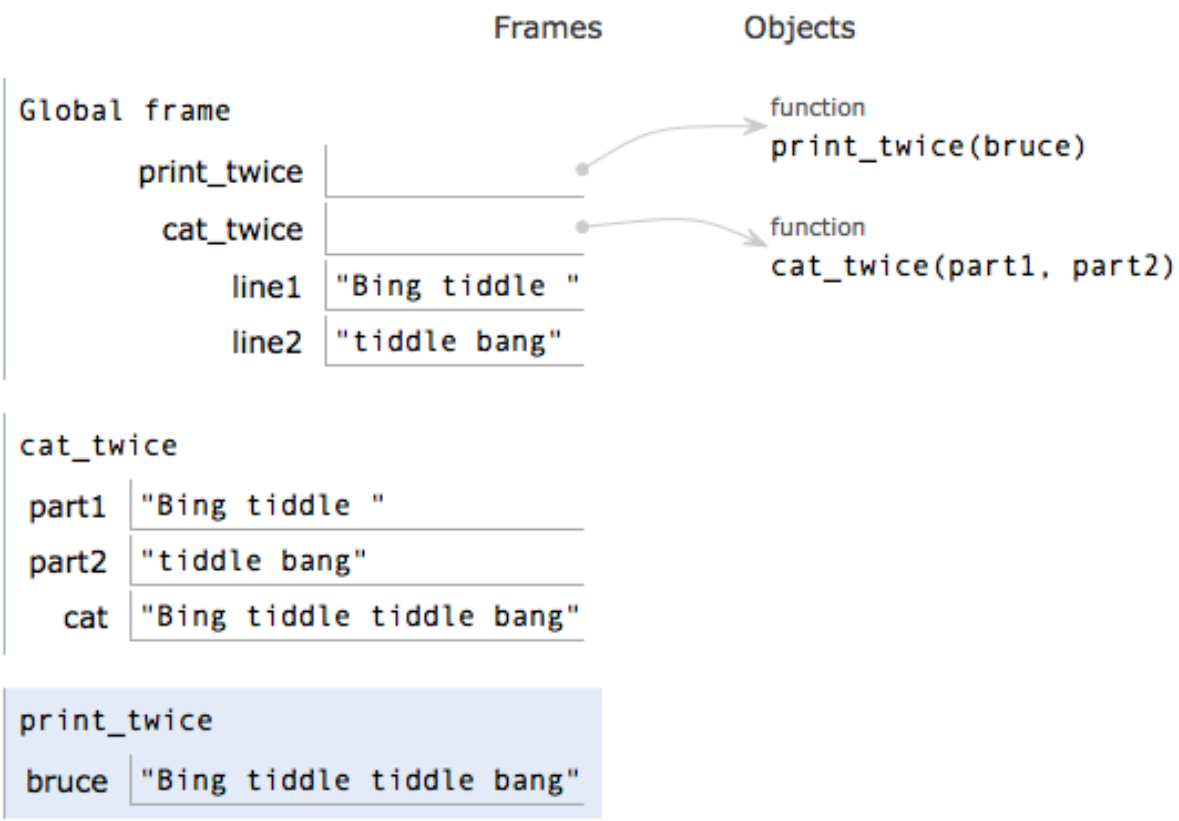
- Parameter sind nur innerhalb der Funktion sichtbar.
- Lokal (durch Zuweisung) eingeführte Variablen ebenfalls.

Python-Interpreter

```
>>> def cat_twice(part1, part2):
...     cat = part1 + part2
...     print_twice(cat)
...
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
>>> cat
NameError: name 'cat' is not defined
```

Stapeldiagramm

- Entsprechend zu den Zustandsdiagrammen kann man die Variablenbelegungen in Stapeldiagrammen visualisieren (hier hilft `pythontutor.com`).
  Innerhalb von `print_twice`:



Traceback

UNI FREIBURG

- Tritt bei der Ausführung einer Funktion ein Fehler auf, z.B. Zugriff auf die nicht vorhandene Variable `cat` in `print_twice`, dann gibt es ein Traceback (entsprechend zu unserem Stapeldiagramm):

Python-Interpreter

```
>>> cat_twice(line1, line2)
```

Traceback

UNI FREIBURG

- Tritt bei der Ausführung einer Funktion ein Fehler auf, z.B. Zugriff auf die nicht vorhandene Variable `cat` in `print_twice`, dann gibt es ein Traceback (entsprechend zu unserem Stapeldiagramm):

Python-Interpreter

```
>>> cat_twice(line1, line2)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 File "<stdin>", line 3, in cat_twice
 File "<stdin>", line 3, in print_twice
NameError: global name 'cat' is not defined
```

Globale Variablen

UNI FREIBURG

- Man sollte nur lokale Variable und Parameter nutzen.
- Man kann lesend auf globale Variablen zugreifen, falls es nicht eine lokale Variable gleichen Namens gibt.
- Manchmal möchte man aber auch globale Variablen ändern (z.B. zur globalen Moduseinstellung oder für Zähler): Schlüsselwort `global`.

Python-Interpreter

```
>>> counter = 0
>>> def inc():
...     global counter
...     counter = counter + 1
...
>>>
```

Globale Variablen

UNI FREIBURG

- Man sollte nur lokale Variable und Parameter nutzen.
- Man kann lesend auf globale Variablen zugreifen, falls es nicht eine lokale Variable gleichen Namens gibt.
- Manchmal möchte man aber auch globale Variablen ändern (z.B. zur globalen Moduseinstellung oder für Zähler): Schlüsselwort `global`.

Python-Interpreter

```
>>> counter = 0
>>> def inc():
...     global counter
...     counter = counter + 1
...
>>> inc()
>>>
```

Globale Variablen

UNI FREIBURG

- Man sollte nur lokale Variable und Parameter nutzen.
- Man kann lesend auf globale Variablen zugreifen, falls es nicht eine lokale Variable gleichen Namens gibt.
- Manchmal möchte man aber auch globale Variablen ändern (z.B. zur globalen Moduseinstellung oder für Zähler): Schlüsselwort `global`.

Python-Interpreter

```
>>> counter = 0
>>> def inc():
...     global counter
...     counter = counter + 1
...
>>> inc()
>>> counter
1
```

UNI FREIBURG

Rückgabewerte

Funktionen mit und ohne Rückgabewert

UNI FREIBURG

- Funktionen können einen Wert zurückgeben, wie z.B. `chr` oder `sin`.
- Einige Funktionen haben keinen Rückgabewert, weil sie nur einen (Seiten-)Effekt verursachen sollen, wie z.B. `inc` und `print`.
- Tatsächlich geben diese den speziellen Wert `None` zurück.

Python-Interpreter

```
>>> result = print('Bruce')
Bruce
>>> result
>>> print(result)
None [≠ der String 'None'!]
```

- `None` ist der einzige Wert des Typs `NoneType`.

Einen Wert zurück geben

UNI FREIBURG

- Soll die Funktion einen Wert zurück geben, müssen wir das Schlüsselwort return benutzen.

Python-Interpreter

```
>>> def sum3(a, b, c):
...     return a + b + c
...
>>> sum3(1, 2, 3)
6
```

return ≠ print

UNI FREIBURG

- Können wir nicht auch `print(·)` benutzen, um einen Funktionswert zurück zu geben?

Python-Interpreter

```
>>> def printsum3(a, b, c):
...     print(a + b + c)
...
>>> sum3(1, 2, 3)
6
```

return ≠ print

UNI FREIBURG

- Können wir nicht auch `print(·)` benutzen, um einen Funktionswert zurück zu geben?

Python-Interpreter

```
>>> def printsum3(a, b, c):
...     print(a + b + c)
...
>>> sum3(1, 2, 3)
6
>>> sum3(1, 2, 3) + 4
```

return ≠ print

UNI FREIBURG

- Können wir nicht auch `print(·)` benutzen, um einen Funktionswert zurück zu geben?

Python-Interpreter

```
>>> def printsum3(a, b, c):
...     print(a + b + c)
...
>>> sum3(1, 2, 3)
6
>>> sum3(1, 2, 3) + 4
10
```

return ≠ print

UNI FREIBURG

- Können wir nicht auch `print(·)` benutzen, um einen Funktionswert zurück zu geben?

Python-Interpreter

```
>>> def printsum3(a, b, c):
...     print(a + b + c)
...
>>> sum3(1, 2, 3)
6
>>> sum3(1, 2, 3) + 4
10
>>> printsum3(1, 2, 3) + 4
```

return $\neq$ print

UNI FREIBURG

- Können wir nicht auch `print(·)` benutzen, um einen Funktionswert zurück zu geben?

Python-Interpreter

```
>>> def printsum3(a, b, c):
...     print(a + b + c)
...
>>> sum3(1, 2, 3)
6
>>> sum3(1, 2, 3) + 4
10
>>> printsum3(1, 2, 3) + 4
6
TypeError: unsupported operand type(s) for +:
'NoneType' and 'int'
```

Zusammenfassung

UNI FREIBURG

- Funktionen sind benannte vorgegebene Programmstücke (Standardfunktionen) oder selbst definierte Funktionen.

Zusammenfassung

UNI FREIBURG

- Funktionen sind benannte vorgegebene Programmstücke (Standardfunktionen) oder selbst definierte Funktionen.
- Beim Aufruf einer Funktion müssen Argumente angegeben werden, die die formalen Parameter mit Werten belegen.

Zusammenfassung

UNI FREIBURG

- Funktionen sind benannte vorgegebene Programmstücke (Standardfunktionen) oder selbst definierte Funktionen.
- Beim Aufruf einer Funktion müssen Argumente angegeben werden, die die formalen Parameter mit Werten belegen.
- Funktionen geben normalerweise einen Funktionswert zurück: `return`.

Zusammenfassung

UNI FREIBURG

- Funktionen sind benannte vorgegebene Programmstücke (Standardfunktionen) oder selbst definierte Funktionen.
- Beim Aufruf einer Funktion müssen Argumente angegeben werden, die die formalen Parameter mit Werten belegen.
- Funktionen geben normalerweise einen Funktionswert zurück: `return`.
- Funktionen führen einen neuen Namensraum ein für die Parameter und lokalen Variablen (durch Zuweisung eingeführt).

Zusammenfassung

UNI FREIBURG

- Funktionen sind benannte vorgegebene Programmstücke (Standardfunktionen) oder selbst definierte Funktionen.
- Beim Aufruf einer Funktion müssen Argumente angegeben werden, die die formalen Parameter mit Werten belegen.
- Funktionen geben normalerweise einen Funktionswert zurück: `return`.
- Funktionen führen einen neuen Namensraum ein für die Parameter und lokalen Variablen (durch Zuweisung eingeführt).
- Lesend kann man immer auf globale Variablen zugreifen, schreibend mit Hilfe des `global`-Schlüsselworts.

Zusammenfassung

UNI FREIBURG

- Funktionen sind benannte vorgegebene Programmstücke (Standardfunktionen) oder selbst definierte Funktionen.
- Beim Aufruf einer Funktion müssen Argumente angegeben werden, die die formalen Parameter mit Werten belegen.
- Funktionen geben normalerweise einen Funktionswert zurück: `return`.
- Funktionen führen einen neuen Namensraum ein für die Parameter und lokalen Variablen (durch Zuweisung eingeführt).
- Lesend kann man immer auf globale Variablen zugreifen, schreibend mit Hilfe des `global`-Schlüsselworts.
- `pythontutor.com` visualisiert die Programmausführung mit Hilfe von Zustands-/Stapeldiagrammen