

Einführung in Python

9. Vorlesung



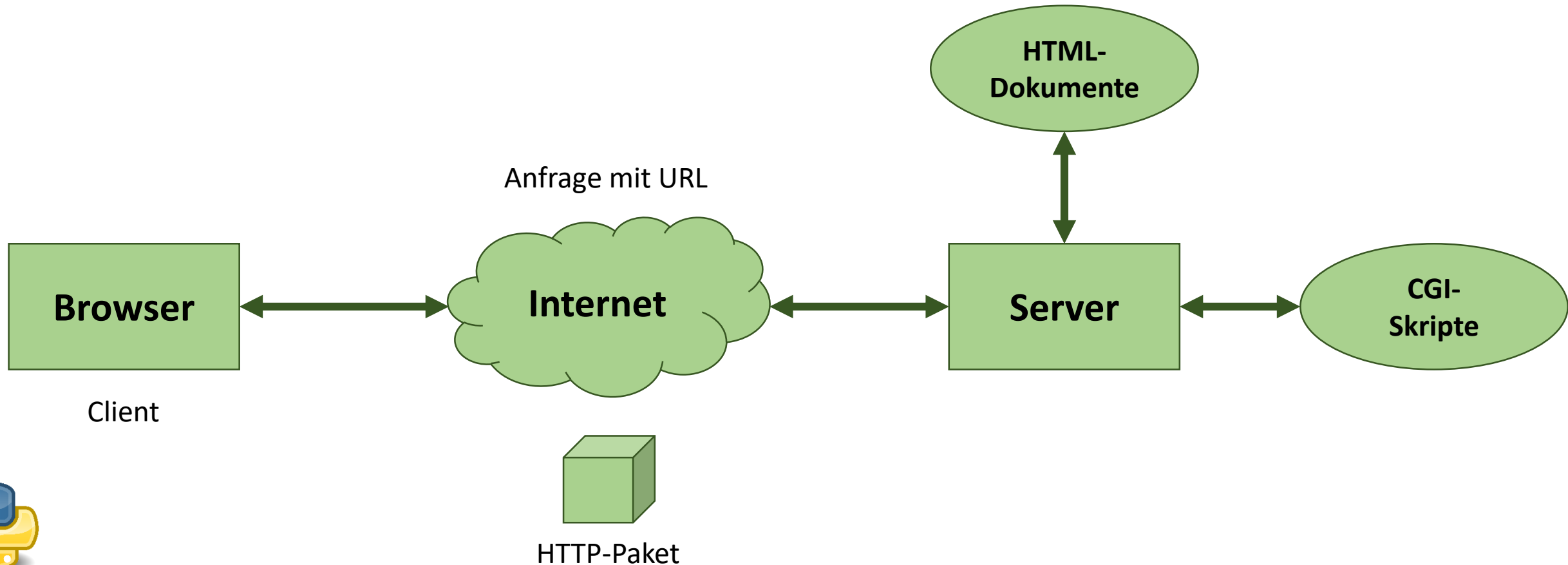
Debugging

Testen & Profiling



Wiederholung letztes Mal

- CGI-Skripte sind Programme die auf einem Server gestartet werden und als Ergebnis ein HTML Dokument erzeugen



Wiederholung letztes Mal

- HTML (Hypertext Markup Language) ist DIE Sprache des Internets

- HTML-Dokumente bestehen aus Elementen, welche wiederum durch so genannte *Tags* repräsentiert werden

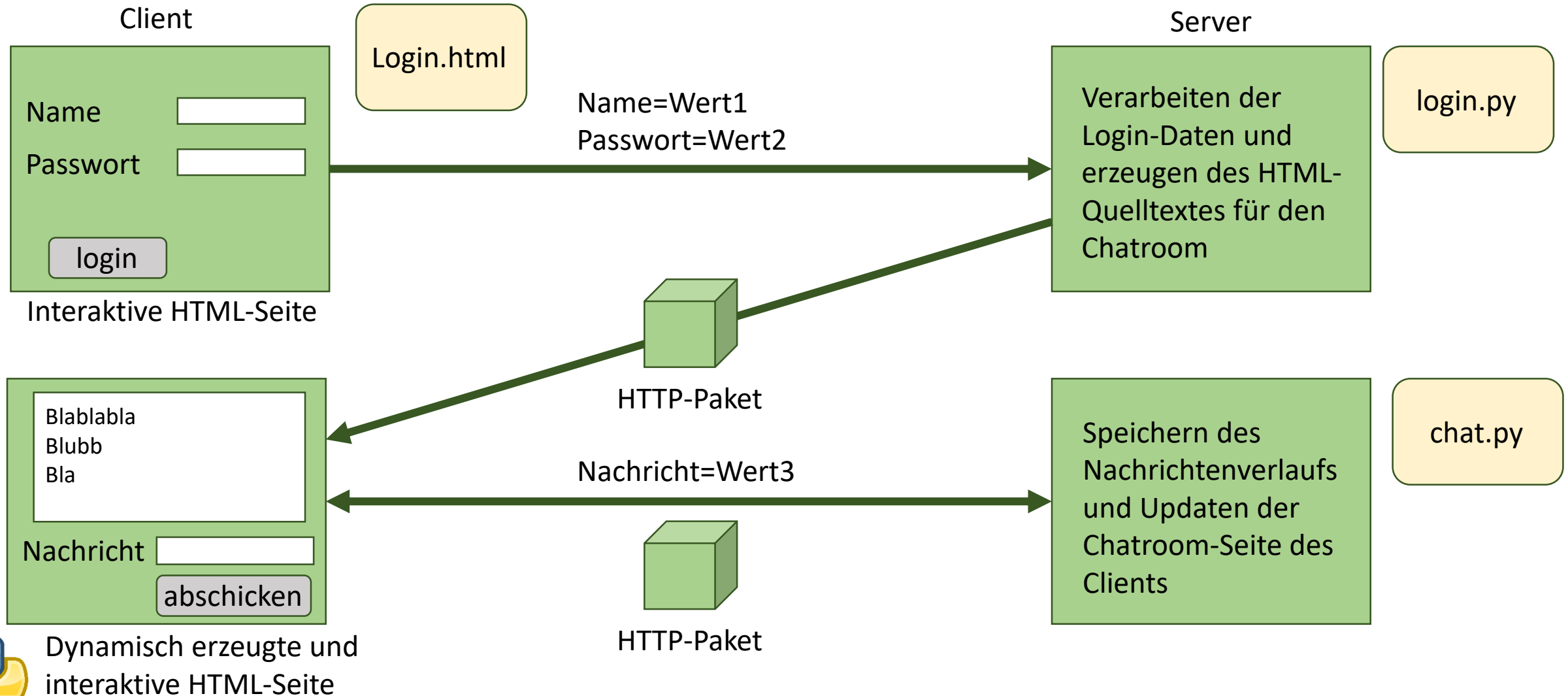
<tagname> Inhalt </tagname>

```
<!DOCTYPE html>
<meta charset="UTF-8">
<html>
<head>
<title> Titel der Webseite </title>
</head>
<body>
    <h1> Überschrift in der Größe 1 </h1>
    <h2> Überschrift in der Größe 2 </h2>
    <p> Ein Paragraph voller Text </p>
    Text <b>ohne</b> Paragraph
    aber mit einem
    <a href="www.python.org"> Hyperlink </a>
    <br>
    <br>
    <image src="./python.png" height="300" weight="300" />

</body>
</html>
```



Wiederholung letztes Mal

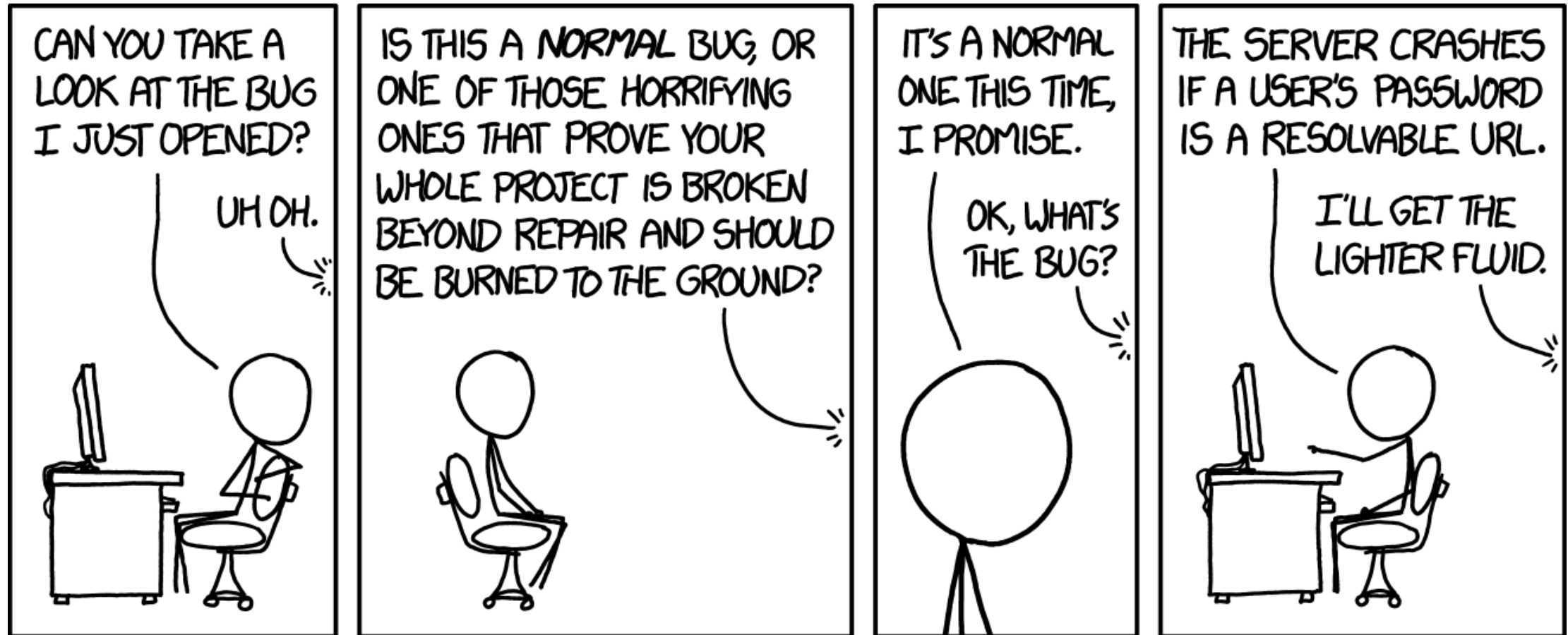


Wiederholung letztes Mal

- Zur Kommunikation von Rechnern über das Internet existieren verschiedene *Protokolle*
- Zu jedem dieser Protokolle gibt es Python-Module, welche Klassen und Funktionen bereitstellen um mit diesen zu arbeiten:
 - Übertragen von Dateien mit FTP
 - Zugriff auf Webseiten mit HTTP (besser: *requests* Modul)
 - E-Mails senden mit SMTP



Debugging – Fehler finden und vermeiden



Debugging – Fehler finden und vermeiden

- Im Grunde gibt es drei Arten von Fehlern:
 - *Syntaxfehler* sind Grammatikfehler die vom Interpreter erkannt und gemeldet werden, **bevor** das Skript ausgeführt wird
 - *Laufzeitfehler* sind Fehler die erst **während** der Ausführung eines Skripts auftreten, wobei der Interpreter eine Exception wirft und eventuell die Ausführung des Programms abbricht (z. B. Division durch Null)
 - *Semantikfehler* liegen vor, wenn das Skript „erfolgreich“ läuft, aber nicht das leistet was es leisten soll



Debugging – Fehler finden und vermeiden

- Im Gegensatz zu Syntax- und Laufzeitfehler kann der Interpreter Semantikfehler nicht erkennen und liefert dementsprechend auch keine Fehlermeldung
- Im Allgemeinen kann man solchen Fehlern folgendermaßen begegnen:
 - Gute und sehr ausführliche Dokumentation des Programmcodes
 - Kontrollen an kritischen Stellen des Programms einbauen, wenn sie nicht erfüllt sind wird ein Programmabbruch ausgelöst (**Exceptions** oder **Asserts**)
 - Starten des Programms in einem speziellem Testmodus, so dass es seine Arbeitsweise dokumentiert (**Logging**)
 - Schritt-für-Schritt Analyse des Programms mit Hilfe eines **Debuggers**



Einfache Beispiele Semantischer Fehler

```
if x > 19:
    if x == y:
        print('Niemand erwartet die')
    else:
        print('spanische Inquisition!')
```

```
if x > 19:
    if x == y:
        print('Niemand erwartet die')
else:
    print('spanische Inquisition!')
```

```
def primzahl_faktoren(zahl):
    fak = [1]
    faktor = 2
    while zahl > 1:
        while zahl % faktor == 0:
            fak.append(faktor)
            zahl /= faktor
            faktor += 1
    return(fak)
```

```
primzahl_faktoren(120)
[1, 2, 2, 2, 3, 5]
```



Einfache Beispiele Semantischer Fehler

```
if x > 19:
    if x == y:
        print('Niemand erwartet die')
    else:
        print('spanische Inquisition!')
```

```
if x > 19:
    if x == y:
        print('Niemand erwartet die')
else:
    print('spanische Inquisition!')
```

```
def primzahl_faktoren(zahl):
    fak = [1]
    faktor = 2
    while zahl > 1:
        while zahl % faktor == 0:
            fak.append(faktor)
            zahl /= faktor
            faktor += 1
    return(fak)
```

```
primzahl_faktoren(19.5)
#Endlos-Schleife
```



Exceptions revisited

- Eine Exception (Ausnahme) ist ein Objekt, welches vom Laufzeitsystem erzeugt wird, wenn das Programm aus bestimmten Gründen abgebrochen werden muss
- Wir wissen: Für verschiedene Fehler gibt es unterschiedliche Ausnahmetypen (*Indexerror, IOError, TypeError, KeyError, ...*)
- Wir wissen: Ausnahmen können mit *try – except (– finally)* abgefangen und verarbeitet werden



Exceptions revisited

- Mit der *raise*-Anweisung kann man gezielt Ausnahmen auslösen (um z. B. Vor- und Nachbedingung zu konstruieren)

raise Exceptionname('Fehlermeldung')

```
def primzahl_faktoren(zahl):  
    if zahl < 0:  
        raise ValueError('Zahl muss >= 0 sein!')  
    if type(zahl) != int:  
        raise TypeError('Zahl muss vom Typ int sein!')  
    fak = [1]  
    faktor = 2  
    while zahl > 1:  
        while zahl % faktor == 0:  
            fak.append(faktor)  
            zahl /= faktor  
            faktor += 1  
    return(fak)
```



Exceptions revisited

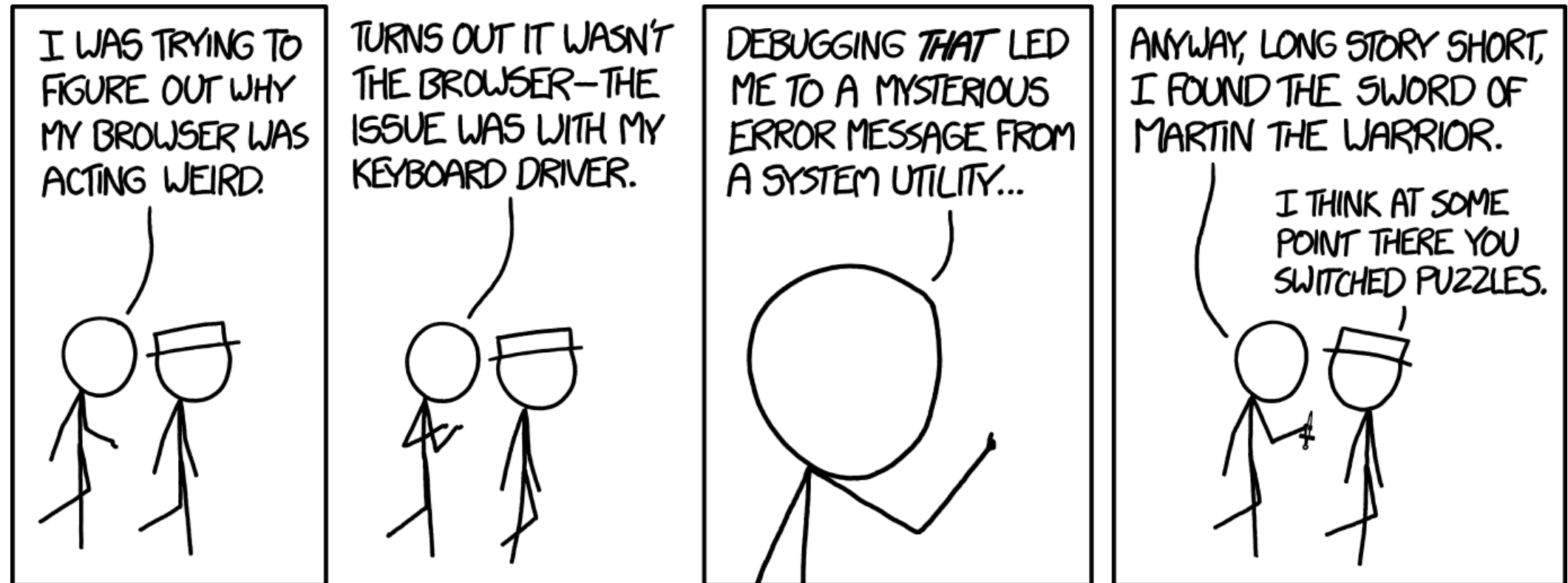
- Es können natürlich auch eigene Exceptions geschrieben werden

```
class primFaktError(ValueError):  
    def __init__(self, arg=('Zahl muss >= 0 sein!',)):  
        self.args = arg
```

```
def primzahl_faktoren(zahl):  
    if zahl < 0:  
        raise primFaktError  
    if type(zahl) != int:  
        raise primFaktError('Zahl muss vom Typ int sein!')  
    fak = [1]  
    faktor = 2  
    while zahl > 1:  
        while zahl % faktor == 0:  
            fak.append(faktor)  
            zahl /= faktor  
            faktor += 1  
    return(fak)
```



Vor- und Nachbedingungen mit *assert*



Vor- und Nachbedingungen mit *assert*

- Strukturierte Pythonprogramme enthalten meist viele Funktions-Methoden- und Klassendefinitionen und diese sollten jeweils für sich unter allen Bedingungen korrekt arbeiten
- Funktionen verarbeiten meist bestimmte Eingabedaten und geben bestimmte Ausgabedaten zurück → An beides sind oft bestimmte Bedingungen geknüpft
- Solche Bedingungen lassen sich mit der *assert*-Funktion überprüfen

assert Bedingung

←
logische Aussage



Vor- und Nachbedingungen mit *assert*

- Ist die Bedingung einer *assert*-Funktion nicht erfüllt, so wird eine *AssertionError* Ausnahme generiert

```
def primzahl_faktoren(zahl):  
    #Prüfe Vorbedingung  
    assert (type(zahl) == int) and (zahl > 0)  
    fak = [1]  
    faktor = 2  
    while zahl > 1:  
        while zahl % faktor == 0:  
            fak.append(faktor)  
            zahl /= faktor  
            faktor += 1  
    #Prüfe Nachbedingung  
    produkt = 1  
    for i in fak:  
        produkt *= i  
    assert produkt == zahl  
    return(fak)
```



Debugging und optimierter Modus

- Das Testen von Vor- und Nachbedingungen kann algorithmisch sehr zeitintensiv und ressourcenaufwändig sein → Verschlechterung der Laufzeit des Programms
- Wird der Pythoninterpreter im optimierten Modus gestartet, so werden alle *assert*-Anweisungen übersprungen

```
user@pc:~$ python3.6 -O meinSkript.py
```



Debugging und optimierter Modus

- Mit der Variablen `__debug__` kann man mehrere Codezeilen vom optimierten Modus ignorieren lassen

```
def primzahl_faktoren(zahl):  
    assert (type(zahl) == int) and (zahl > 0)  
    fak = [1]  
    faktor = 2  
    while zahl > 1:  
        while zahl % faktor == 0:  
            fak.append(faktor)  
            zahl /= faktor  
            faktor += 1  
    #Prüfe Nachbedingung  
    if __debug__:  
        produkt = 1  
        for i in fak:  
            produkt *= i  
        assert produkt == zahl  
    return(fak)
```



Debugging und optimierter Modus

- Man sollte die *assert*-Funktion **niemals** zur Validierung von Eingabedaten nutzen (da sich diese Überprüfung im optimierten Modus umgehen lässt)
- Man kann Ausversehens leicht *assert* Bedingungen formulieren die immer wahr sind (damit sind sie nutzlos)

```
assert(1 == 2, 'Das sollte eine AssertionError Ausnahme erzeugen')
```

- Offensichtliche Fälle erkennt der Pythoninterpreter und gibt eine Warnung aus

```
<input>:1: SyntaxWarning: assertion is always true, perhaps remove parentheses?
```



Logging und Log-Files

ISSUE:

RECENT UPDATE BROKE
SUPPORT FOR HARDWARE
I NEED FOR MY JOB.

WORKAROUND:

IF WE WAIT LONG ENOUGH,
THE EARTH WILL EVENTUALLY
BE CONSUMED BY THE SUN.



Selbstdokumentation

- Eine beliebte Form des Fehlerfindens ist die Selbstdokumentierung eines Programms mithilfe der *print*-Anweisung

```
def quicksort(liste):  
    if len(liste) > 0:  
        print(f'Ich sortiere {liste}')  
        print(f'Element zum Spalten {liste[0]}')  
    if len(liste) <= 1:  
        return(liste)  
    else:  
        return(quicksort([x for x in liste[1:] if x < liste[0]])\  
                +[liste[0]]\  
                + quicksort([x for x in liste[1:] if x >= liste[0]]))
```

```
>>> quicksort([149, 36, 5, 69, 19, 119])  
Ich sortiere [149, 36, 5, 69, 19, 119]  
Element zum Spalten 149  
Ich sortiere [36, 5, 69, 19, 119]  
Element zum Spalten 36  
Ich sortiere [5, 19]
```



Selbstdokumentation

- Eine beliebte Form des Fehlerfindens ist die Selbstdokumentierung eines Programms mithilfe der *print*-Anweisung

```
def quicksort(liste):  
    if len(liste) > 0:  
        print(f'Ich sortiere {liste}')  
        print(f'Element zum Spalten {liste[0]}')  
        if len(liste) <= 1:  
            return(liste)  
        else:  
            return(quicksort([x for x in liste[1:] if x < liste[0]])\  
                    +[liste[0]]\  
                    + quicksort([x for x in liste[1:] if x >= liste[0]]))
```

```
Element zum Spalten 5  
Ich sortiere [19]  
Element zum Spalten 19  
Ich sortiere [69, 119]  
Element zum Spalten 69  
Ich sortiere [119]  
Element zum Spalten 119  
[5, 19, 36, 69, 119, 149]
```



Dokumentation mit Log-Dateien

- Eine Alternative zur Dokumentation mittels der *print*-Funktion ist es den aktuellen Stand eines Programms in eine Textdatei zu schreiben und dieses hinterher zu analysieren
- Anstatt eines eigenen File-Handles kann man die Funktionen des Moduls *logging* verwenden
- Mit diesem kann man verschiedene Logger erzeugen und verwalten und auf unterschiedliche Logging-Level schreiben lassen



Dokumentation mit Log-Dateien

```
>>> import logging  
  
>>> logging.basicConfig(filename='/tmp/logFile.txt', level=logging.DEBUG)  
>>> logging.debug('Erster Eintrag')  
>>> x = 19  
>>> logging.debug(x)  
>>> logging.info('Zweiter Eintrag')
```

logFile.txt

```
DEBUG:root:Erster Eintrag  
DEBUG:root:19  
INFO:root:Zweiter Eintrag
```



Dokumentation mit Log-Dateien

- Es gibt verschiedene Logging-Level um die Dringlichkeit von Einträgen zu definieren
- Durch die Einstellung des Logging-Levels können unwichtige Meldungen unterdrückt werden



Beispiel Logging-Level

```
logging.basicConfig(filename='/tmp/logFile.txt', level=logging.DEBUG, filemode='w')
```

```
def merge(liste1, liste2):  
    logging.debug(f' Starte merge({liste1},{liste2})')  
    if len(liste1) == 0: ergebnis = liste2  
    elif len(liste2) == 0: ergebnis = liste1  
    else:  
        a, b = liste1[0], liste2[0]  
        if a <= b: ergebnis = [a] + merge(liste1[1:] + liste2)  
        else: ergebnis = [b] + merge(liste1 + liste2[1:])  
    logging.debug(f'Ergebnis von merge({liste1},{liste2}): {ergebnis}')  
    return(ergebnis)
```

```
def merge_sort(liste):  
    logging.info(f'merge_sort({liste})')  
    if len(liste) <= 1: ergebnis = liste  
    else:  
        liste1, liste2 = liste[:len(liste)//2], liste[len(liste)//2:]  
        ergebnis = merge(merge_sort(liste1), merge_sort(liste2))  
    logging.info(f'Ergebnis von merge_sort({liste}): {ergebnis}')  
    return(ergebnis)
```



Beispiel Logging-Level

```
merge_sort([149, 36, 5, 119, 69, 19, 17, 67, 117, 3, 34, 147])
```

```
INFO:root:merge_sort([149, 36, 5, 119, 69, 19, 17, 67, 117, 3, 34, 147])
INFO:root:merge_sort([149, 36, 5, 119, 69, 19])
INFO:root:merge_sort([149, 36, 5])
INFO:root:merge_sort([149])
INFO:root:Ergebnis von merge_sort([149]): [149]
INFO:root:merge_sort([36, 5])
INFO:root:merge_sort([36])
INFO:root:Ergebnis von merge_sort([36]): [36]
INFO:root:merge_sort([5])
INFO:root:Ergebnis von merge_sort([5]): [5]
DEBUG:root: Starte merge([36],[5])
DEBUG:root: Starte merge([36],[])
DEBUG:root:Ergebnis von merge([36],[]): [36]
DEBUG:root:Ergebnis von merge([36],[5]): [5, 36]
INFO:root:Ergebnis von merge_sort([36, 5]): [5, 36]
DEBUG:root: Starte merge([149],[5, 36])
DEBUG:root: Starte merge([149],[36])
DEBUG:root: Starte merge([149],[])
DEBUG:root:Ergebnis von merge([149],[]): [149]
DEBUG:root:Ergebnis von merge([149],[36]): [36, 149]
DEBUG:root:Ergebnis von merge([149],[5, 36]): [5, 36, 149]
INFO:root:Ergebnis von merge_sort([149, 36, 5]): [5, 36, 149]
INFO:root:merge_sort([119, 69, 19])
INFO:root:merge_sort([119])
...
```



Beispiel Logging-Level

```
logging.basicConfig(filename='/tmp/logFile.txt', level=logging.INFO, filemode='w')
#Das logging wurde auf das höhere Level INFO gesetzt
def merge(liste1, liste2):
    logging.debug(f' Starte merge({liste1},{liste2})')
    if len(liste1) == 0: ergebnis = liste2
    elif len(liste2) == 0: ergebnis = liste1
    else:
        a, b = liste1[0], liste2[0]
        if a <= b: ergebnis = [a] + merge(liste1[1:] + liste2)
        else: ergebnis = [b] + merge(liste1 + liste2[1:])
    logging.debug(f'Ergebnis von merge({liste1},{liste2}): {ergebnis}')
    return(ergebnis)

def merge_sort(liste):
    logging.info(f'merge_sort({liste})')
    if len(liste) <= 1: ergebnis = liste
    else:
        liste1, liste2 = liste[:len(liste)//2], liste[len(liste)//2:]
        ergebnis = merge(merge_sort(liste1), merge_sort(liste2))
    logging.info(f'Ergebnis von merge_sort({liste}): {ergebnis}')
    return(ergebnis)
```



Beispiel Logging-Level

```
merge_sort([149, 36, 5, 119, 69, 19, 17, 67, 117, 3, 34, 147])
```

```
INFO:root:merge_sort([149, 36, 5, 119, 69, 19, 17, 67, 117, 3, 34, 147])
INFO:root:merge_sort([149, 36, 5, 119, 69, 19])
INFO:root:merge_sort([149, 36, 5])
INFO:root:merge_sort([149])
INFO:root:Ergebnis von merge_sort([149]): [149]
INFO:root:merge_sort([36, 5])
INFO:root:merge_sort([36])
INFO:root:Ergebnis von merge_sort([36]): [36]
INFO:root:merge_sort([5])
INFO:root:Ergebnis von merge_sort([5]): [5]
INFO:root:Ergebnis von merge_sort([36, 5]): [5, 36]
INFO:root:Ergebnis von merge_sort([149, 36, 5]): [5, 36, 149]
INFO:root:merge_sort([119, 69, 19])
INFO:root:merge_sort([119])
INFO:root:Ergebnis von merge_sort([119]): [119]
INFO:root:merge_sort([69, 19])
INFO:root:merge_sort([69])
INFO:root:Ergebnis von merge_sort([69]): [69]
INFO:root:merge_sort([19])
INFO:root:Ergebnis von merge_sort([19]): [19]
INFO:root:Ergebnis von merge_sort([69, 19]): [19, 69]
INFO:root:Ergebnis von merge_sort([119, 69, 19]): [19, 69, 119]
INFO:root:Ergebnis von merge_sort([149, 36, 5, 119, 69, 19]): [5, 19, 36, 69, 119, 149]
INFO:root:merge_sort([17, 67, 117, 3, 34, 147])
...
```



Logger-Objekte

- Für große Projekte eignet es sich mehrere Logger zu verwenden, welchen auch unterschiedliche Logging-Level zugeordnet sein können

```
import logging

logging.basicConfig(filename='/tmp/logFile.txt')
log1 = logging.getLogger('Modul 1')           #Erzeugen eines Logger-Objekts
log1.setLevel(logging.INFO)

log2 = logging.getLogger('Modul 2')           #Erzeugen eines Logger-Objekts
log2.setLevel(logging.DEBUG)

log1.debug('Unwichtige Meldung')
log1.info('wichtige Meldung')
log2.debug('wichtige Meldung')
```

```
INFO:Modul 1:wichtige Meldung
DEBUG:Modul 2:wichtige Meldung
```



Logging-Meldungen konfigurieren

- Beim Aufruf eines Loggers mittels *logging.basicConfig()* kann man mit Hilfe des optionalen Parameters *format* das Format der Meldungen ändern
- Erwartet wird ein Formatstring, welcher (unter Anderen) aus den folgenden Platzhaltern aufgebaut sein kann:

Platzhalter	Bedeutung
%(asctime)s	Zeitpunkt zu dem die Meldung ausgegeben wurde.
%(funcname)s	Name der Funktion, in der die Meldung abgesetzt wurde.
%(levelname)s	Logging-Level.
%(message)s	Text der Meldung.
%(module)s	Name des Moduls, in dem die Meldung abgesetzt wurde.
%(name)s	Name des Loggers.
%(pathname)s	Vollständiger Pfad des Moduls, in dem die Meldung abgesetzt wurde.



Logging-Meldungen konfigurieren

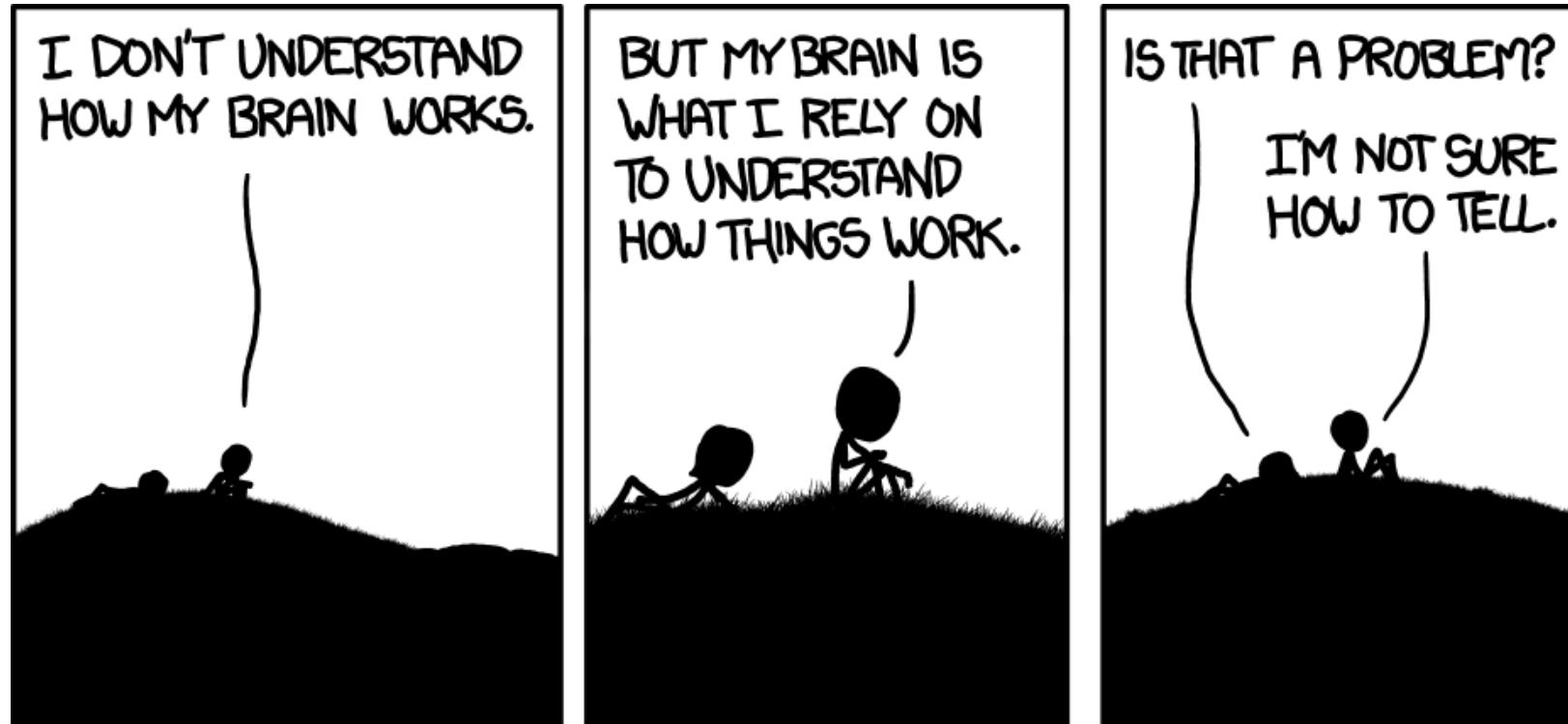
```
>>> import logging

>>> fstring = '%(asctime)s:::%(name)s:::%(funcName)s:::%(message)s'
>>> logging.basicConfig(filename='/tmp/logFile.txt', format=fstring)
>>> logging.critical('Super wichtige Meldung!')
>>> def tolle_funktion():
>>>     logging.critical('Noch eine wichtige Meldung!')
>>> tolle_funktion()
>>> log1 = logging.getLogger('Logger-Droide')
>>> log1.critical('Roger Roger')
```

```
2016-01-24 21:45:16,080:::root:::<module>:::Super wichtige Meldung!
2016-01-24 21:46:26,176:::root:::tolle_funktion:::Noch eine wichtige Meldung!
2016-01-24 21:48:16,025:::Logger-Droide:::<module>:::Roger Roger!
```

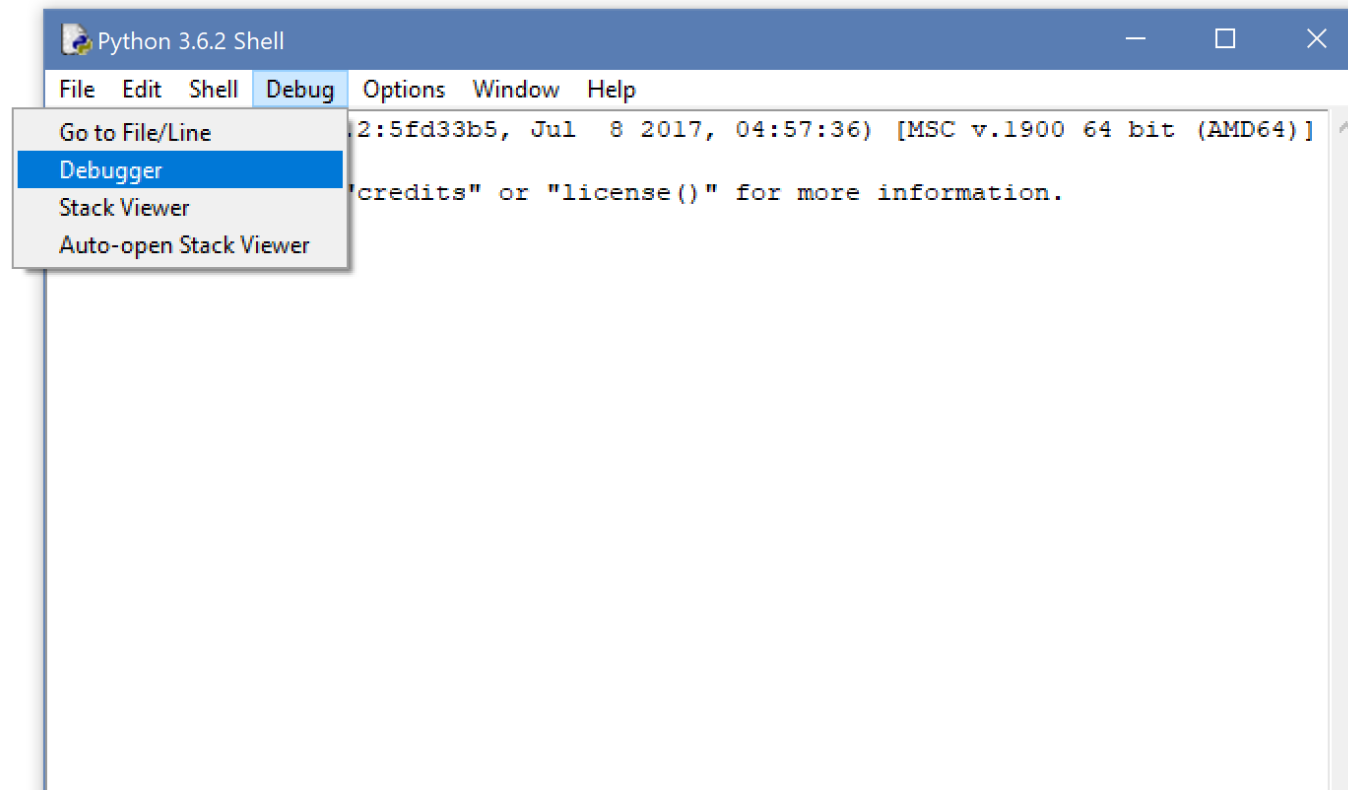


Der IDLE Debugger



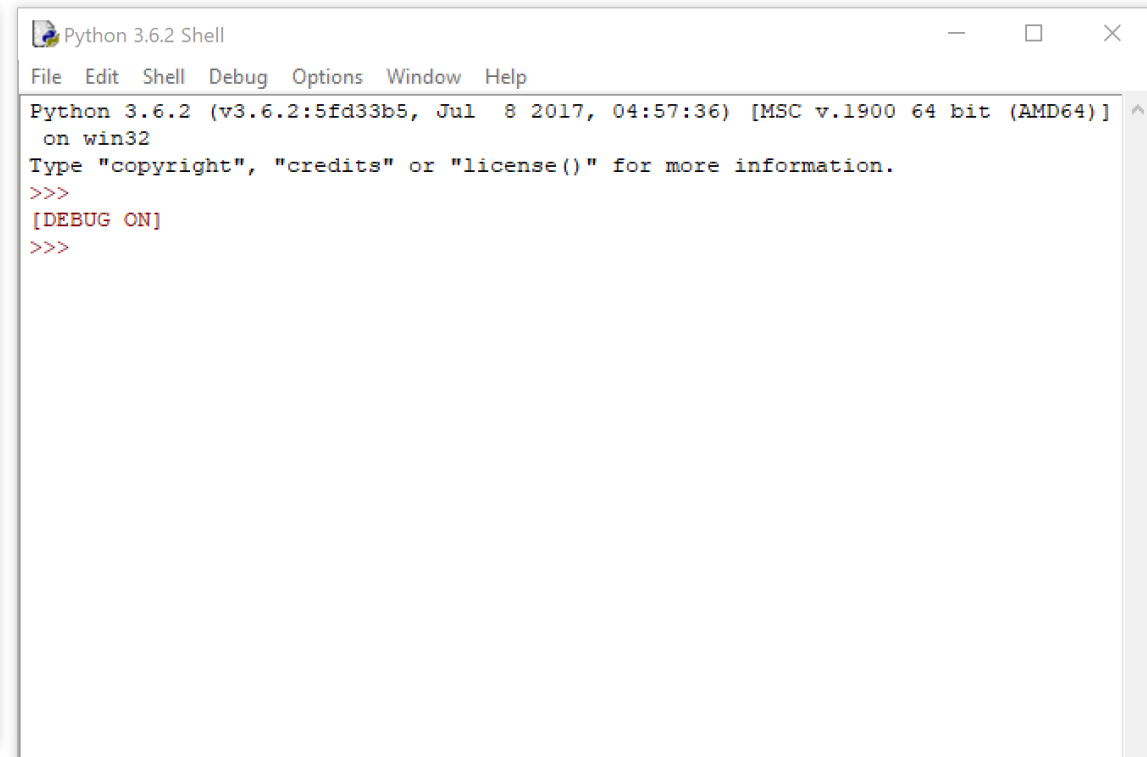
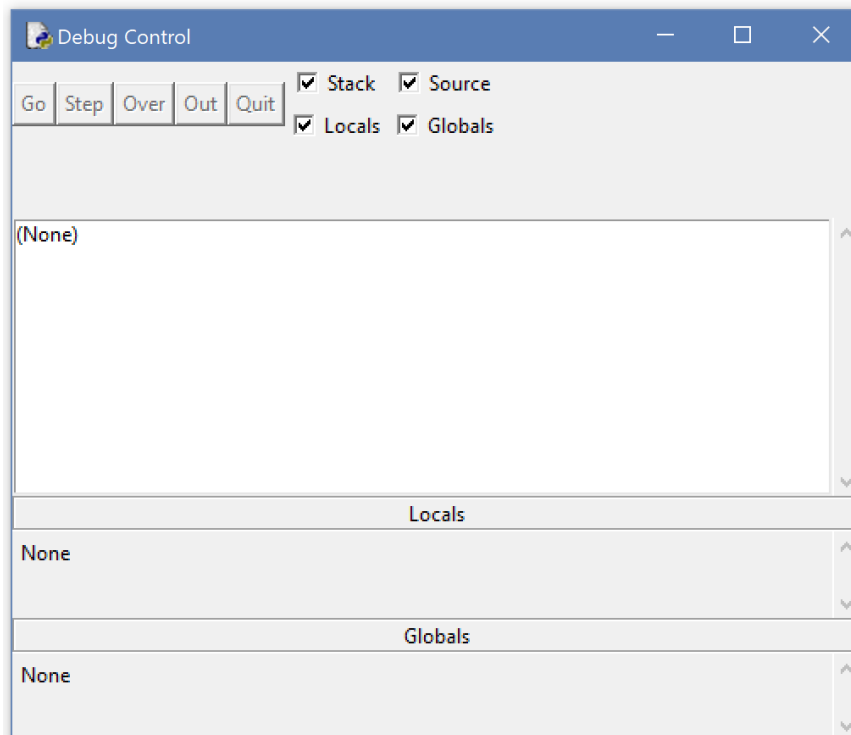
Der IDLE Debugger

- Die Python IDLE-Umgebung besitzt einen integrierten Debugger, welchen man folgendermaßen aufruft:
 - In einem Shellfenster aktiviert man den Debugger über das Menü *Debug*



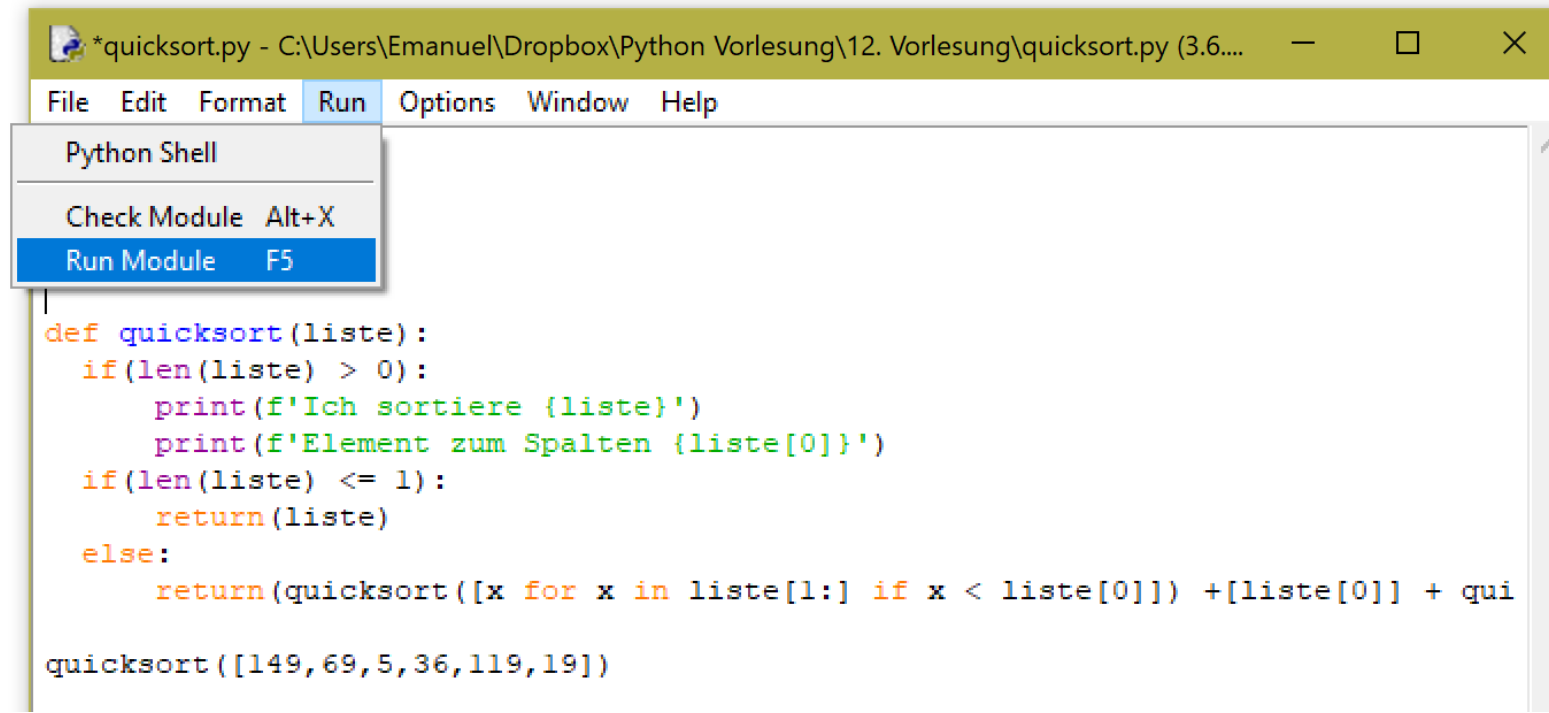
Der IDLE Debugger

- Die Python IDLE-Umgebung besitzt einen integrierten Debugger, welchen man folgendermaßen aufruft:
 - In einem Shellfenster aktiviert man den Debugger über das Menü *Debug*



Der IDLE Debugger

- Die Python IDLE-Umgebung besitzt einen integrierten Debugger, welchen man folgendermaßen aufruft:
 - Startet man nun im Editor-Fenster das Skript mittels *Run Modul*, so läuft das Programm unter der Kontrolle des Debuggers



```
*quicksort.py - C:\Users\Emanuel\Dropbox\Python Vorlesung\12. Vorlesung\quicksort.py (3.6...
File Edit Format Run Options Window Help
Python Shell
Check Module Alt+X
Run Module F5

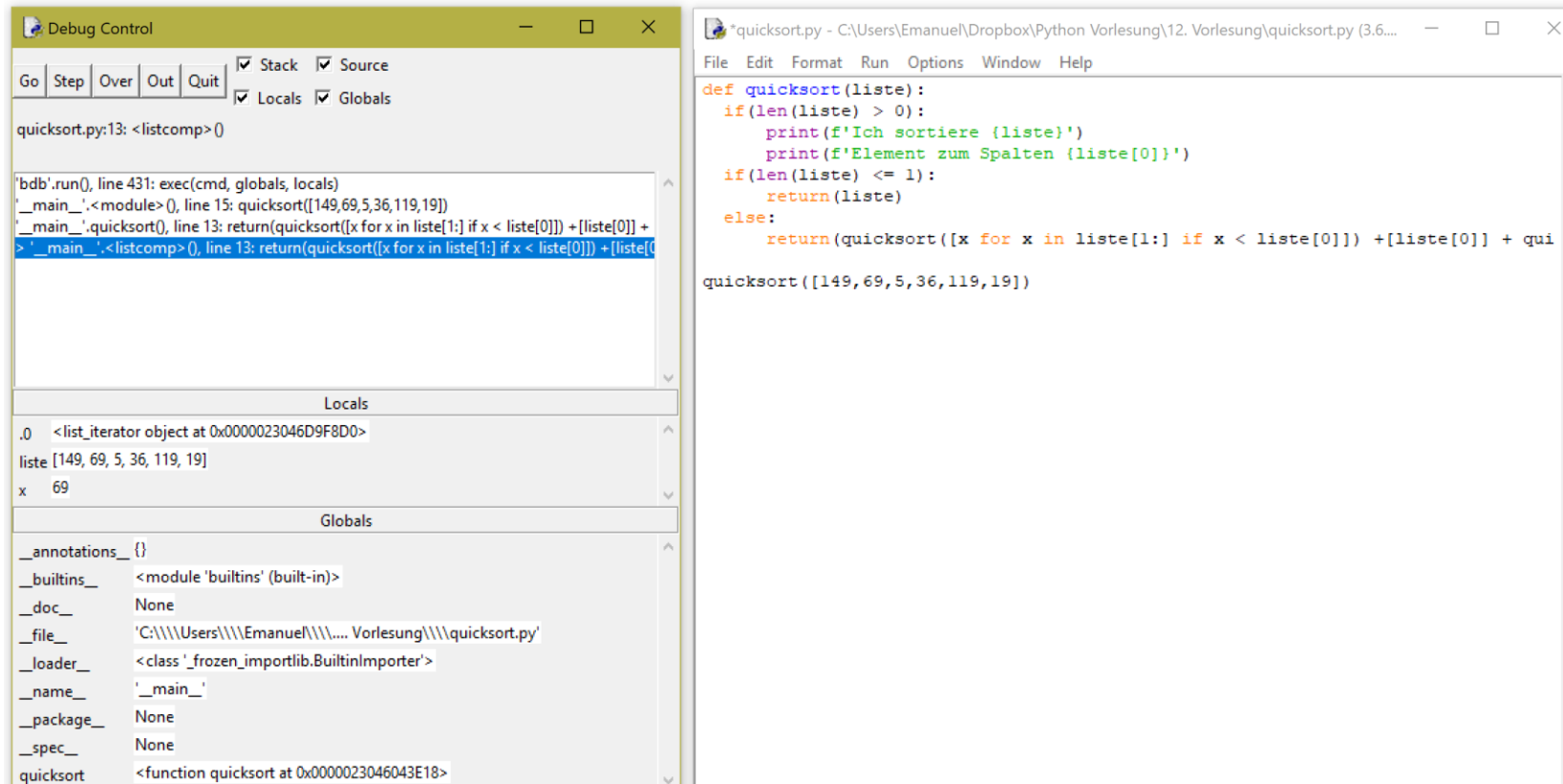
def quicksort(liste):
    if(len(liste) > 0):
        print(f'Ich sortiere {liste}')
        print(f'Element zum Spalten {liste[0]}')
    if(len(liste) <= 1):
        return(liste)
    else:
        return(quicksort([x for x in liste[1:] if x < liste[0]]) +[liste[0]] + qui

quicksort([149,69,5,36,119,19])
```



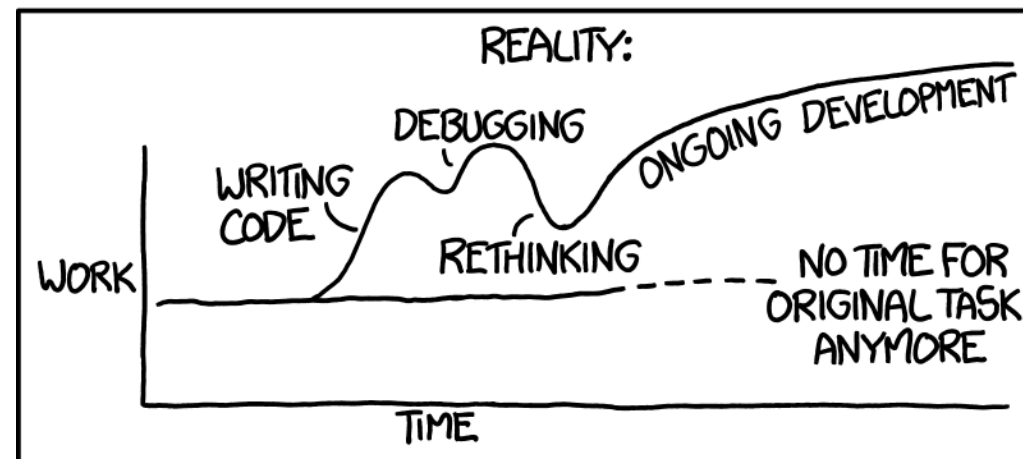
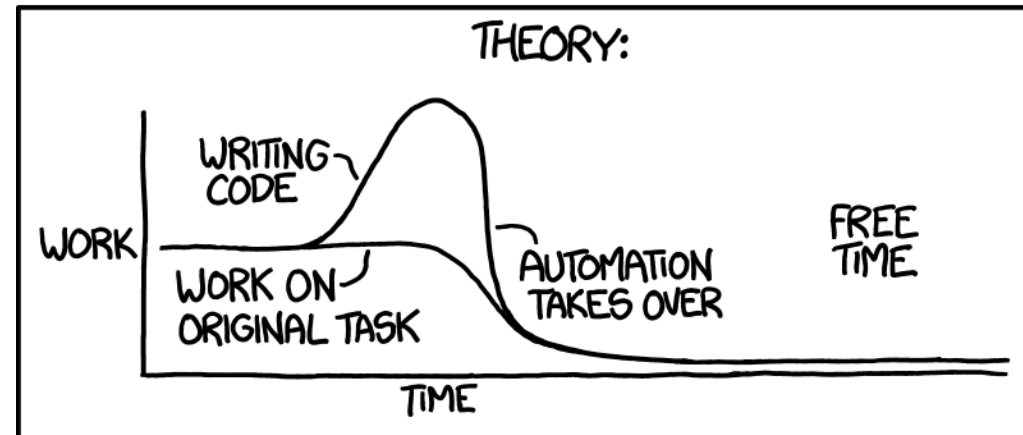
Der IDLE Debugger

- Die Python IDLE-Umgebung besitzt einen integrierten Debugger, welchen man folgendermaßen aufruft:
 - Startet man nun im Editor-Fenster das Skript mittels *Run Modul*, so läuft das Programm unter der Kontrolle des Debuggers



Testen und Tuning

"I SPEND A LOT OF TIME ON THIS TASK.
I SHOULD WRITE A PROGRAM AUTOMATING IT!"



Automatisiertes Testen mit *doctest*

- Das Modul *doctest* ermöglicht es automatisches Tests mittels so genannter *Docstrings* auszuführen
- Ein *Docstring* ist ein langer String, welcher meist zu Beginn eines Moduls oder unterhalb der Kopfzeile einer Funktion steht
- Er wird nicht als Code sondern als Kommentar interpretiert



Doctest Beispiel

```
def quadratsumme(liste):  
    """ Summer der Quadrate der Elemente einer Zahlenliste  
    >>> quadratsumme([1,2,3])  
    14  
    >>> quadratsumme([10,10,10])  
    300  
    >>> quadratsumme([])  
    0  
    """  
  
    summe = 0  
    for x in liste:  
        summe += x**2  
    return(summe)
```



Doctest Beispiel

- Der *Docstring* enthält Testaufrufe der Funktion, d. h. man tut so als würde man die Funktion im interaktivem Modus testen
- Zeilen die mit dem Python-Prompt „>>>“ starten werden interpretiert, darauf folgende Zeilen ohne Prompt stellen das richtige Ergebnis des Aufrufs dar

```
def quadratsumme(liste):  
    """ Summer der Quadrate der Elemente einer Zahlenliste  
    >>> quadratsumme([1,2,3])  
    14  
    >>> quadratsumme([10,10,10])  
    300  
    >>> quadratsumme([])  
    0  
    """  
  
    summe = 0  
    for x in liste:  
        summe += x**2  
    return(summe)
```



Doctest Beispiel

- *Doctests* startet man entweder über das Modul *doctest*

```
import doctest
doctest.testmode(verbose=True)
```

```
Trying:
quadratsumme([1,2,3])
Expecting:
14
ok
Trying:
quadratsumme([10,10,10])
Expecting:
300
ok
Trying:
quadratsumme([])
Expecting:
0
ok
1 items had no tests:
__main__
1 items passed all tests:
3 tests in __main__.quadratsumme
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
TestResults(failed=0, attempted=3)
```



Doctest Beispiel

- Oder über die Kommandozeile mit dem Python-Interpreter

```
user@pc:~$ python3.6 -m doctest -v quadratsummen.py
```

```
Trying:
quadratsumme([1,2,3])
Expecting:
14
ok
Trying:
quadratsumme([10,10,10])
Expecting:
300
ok
Trying:
quadratsumme([])
Expecting:
0
ok
1 items had no tests:
__main__
1 items passed all tests:
3 tests in __main__.quadratsumme
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
```



Doctest Beispiel

- *DocStrings* können auch über mehrere Zeilen interpretiert werden

```
def sinnlose_Funktion():  
    """ >>> # Kommentare werden auch ignoriert  
    >>> x = 19  
    >>> if(x == 149):  
    ...     print('Alles richtig gemacht!')  
    ... else:  
    ...     print('Oh nein!')  
    ...     print('Alles schief gelaufen!')  
    Oh nein!  
    Alles schief gelaufen!  
    """  
  
    pass
```



Doctest Beispiel

- *DocStrings* können auch in separaten Textdateien liegen

sinnlose_funktion_test.txt

```
>>> # Kommentare werden auch ignoriert
>>> x = 19
>>> if(x == 149):
...     print('Alles richtig gemacht!')
... else:
...     print('Oh nein!')
...     print('Alles schief gelaufen!')
Oh nein!
Alles schief gelaufen!
```

```
user@pc:~$ python3.6 -m doctest -v sinnlose_funktion_test.txt
```

```
import doctest
doctest.testfile('/home/scripts/sinnlose_funktion_test.txt', verbose=True)
```



Automatisiertes Testen mit *unittest*

- Abgeleitet von den Unit Tests aus Java gibt es in Python das Modul *unittest* zum erstellen von Testklassen und dazugehörigen Testreihen
- Innerhalb einer abgeleiteten Testklasse werden (relevante) Tests in Form von Methoden definiert
- Somit lassen sich generalisierte, automatische Testroutinen schreiben



Unittest Beispiel

```
def fibonacci(n):  
    if n == 0:  
        ergebnis = 0  
    elif n == 1:  
        ergebnis = 1  
    else:  
        ergebnis = fibonacci(n-1) + fibonacci(n-2)  
  
    return(ergebnis)
```



Unittest Beispiel

```
import unittest

class testFibonacci(unittest.TestCase):
    def setUp(self):
        #Geerbte Methode zum initiieren von Testdaten
        self.testZahlen = [1,2,5,25,35]

    def testElementarfall(self):
        self.assertEqual(fibonacci(self.testZahlen[0]), 1)

    def testEinfacheFall(self):
        self.assertEqual(fibonacci(self.testZahlen[1]), 1)

    def testDreiSchwereFälle(self):
        self.assertEqual(fibonacci(self.testZahlen[2]), 5)
        self.assertEqual(fibonacci(self.testZahlen[3]), 75025)
        self.assertEqual(fibonacci(self.testZahlen[4]), 9227465)
```



Unittest Beispiel

- Man kann mit einem *TestRunner* einzelne Test-Objekte auswerten lassen

```
>>> test1 = testFibonacci('testElementarfall')
>>> testrunner = unittest.TextTestRunner(verbosity=1)
>>> testrunner.run(test1)
testElementarfall (__main__.testFibonacci) ... ok
-----
Ran 1 test in 0.001s
OK
<unittest.runner.TextTestResult run=1 errors=0 failures=0>
```



Unittest Beispiel

- Oder mit einer *TestSuite* gleich mehrere Tests auf einmal

```
>>> suite = unittest.TestSuite()
>>> suite.addTests([testFibonacci('testElementarfall'),\
                    testFibonacci('testEinfacheFall'),\
                    testFibonacci('testDreiSchwereFälle')])
>>> testrunner = unittest.TextTestRunner(verbosity=2)
>>> testrunner.run(suite)
testElementarfall (__main__.testFibonacci) ... ok
testEinfacheFall (__main__.testFibonacci) ... ok
testDreiSchwereFälle (__main__.testFibonacci) ... ok
-----
Ran 3 tests in 6.065s
OK
<unittest.runner.TextTestResult run=3 errors=0 failures=0>
```



Unittest Beispiel

```
import unittest

def fibonacci_falsch(n):

    if n == 0:
        ergebnis = 0
    elif n == 1:
        ergebnis = 1
    else:
        ergebnis = fibonacci(n-1) + fibonacci(n-1) #Fehler in der Berechnung

    return(ergebnis)

class testFiboFalsch(unittest.TestCase):
    #Wenn man keine Testdaten initiieren muss,
    #braucht man die setUp()-Methode auch nicht überschreiben
    def testFiboBerechnung(self):
        self.assertEqual(fibonacci_falsch(5), 5)
```



Unittest Beispiel

```
>>> test2 = testFiboFalsch('testFiboBerechnung')
>>> testrunner = unittest.TextTestRunner(verbosity=1)
>>> testrunner.run(test2)

=====
FAIL: testFiboBerechnung (__main__.testFiboFalsch)
-----
Traceback (most recent call last):
  File "<ipython-input-72-420925e57017>", line 5, in testFiboBerechnung
    self.assertEqual(fibonacci_falsch(5), 5)
AssertionError: 6 != 5
-----

Ran 1 test in 0.001s
FAILED (failures=1)
<unittest.runner.TextTestResult run=1 errors=0 failures=1>
```



Unittest Prüfmethoden

- Neben *assertEqual* besitzt die Klasse *TestCase* noch weitere Prüfmethoden, unter Anderem:

Methode	Bedeutung
<code>assertEqual(X, Y)</code>	Schlägt fehl wenn X und Y nicht Wertegleich sind.
<code>assertAlmostEqual(X, Y, Z)</code>	Schlägt fehl wenn X und Y bis auf Z Stellen genau gleich sind.
<code>assertDictEqual(X, Y)</code>	Schlägt fehl wenn X und Y ungleiche Dictionaries sind.
<code>assertTrue(X)</code>	Schlägt fehl wenn der Ausdruck X nicht True ist.
<code>assertGreater(X, Y)</code>	Schlägt fehl wenn X kleiner oder gleich Y ist.
<code>assertIn(X, Y)</code>	Schlägt fehl wenn X nicht in Y enthalten ist.
<code>assertMultiLineEqual(X, Y)</code>	Mehrzeilige Strings werden auf Gleichheit geprüft.
<code>assertIs(X, Y)</code>	Schlägt fehl wenn X nicht identisch mit Y ist.
<code>assertNotRegexMatches(X, Y)</code>	Schlägt fehl wenn der Text X nicht auf den regulären Ausdruck Y passt.



Performanz messen mit *profile*

- Übergibt man der *run()*-Methode des Moduls *profiler* eine beliebige Pythonanweisung als String, so erhält man einen Laufzeitperformanzbericht über diese Anweisung

```
def sort(liste):  
    unsortiert = liste[:]   
    sortiert = []  
    while unsortiert:  
        x = min(unsortiert)  
        sortiert.append(x)  
        unsortiert.remove(x)  
    return(sortiert)
```



Performanz messen mit *profile*

```
import profile, random
```

```
liste = [random.randint(0,1000) for i in range(10000)]  
profile.run('sort(liste)')
```

30005 function calls in 1.891 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
10000	0.000	0.000	0.000	0.000	:0(append)
1	0.000	0.000	1.891	1.891	:0(exec)
10000	1.266	0.000	1.266	0.000	:0(min)
10000	0.562	0.000	0.562	0.000	:0(remove)
1	0.000	0.000	0.000	0.000	:0(setprofile)
1	0.062	0.062	1.891	1.891	<input>:1(sort)
1	0.000	0.000	1.891	1.891	<string>:1(<module>)
0	0.000		0.000		profile:0(profiler)
1	0.000	0.000	1.891	1.891	profile:0(sort(liste))



Performanz messen mit *profile*

```
import profile, random

profile.run('[sort(x) for x in [[random.randint(0,1000) for i in range(10000)]
for y in range(100)]]')
```

8640192 function calls (8640092 primitive calls) in 150.125 seconds
Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1000000	1.984	0.000	1.984	0.000	:0(append)
1000000	2.562	0.000	2.562	0.000	:0(bit_length)
1	0.000	0.000	150.125	150.125	:0(exec)
1639986	4.031	0.000	4.031	0.000	:0(getrandbits)
1000000	78.828	0.000	78.828	0.000	:0(min)
1000000	33.062	0.000	33.062	0.000	:0(remove)
1	0.000	0.000	0.000	0.000	:0(setprofile)
100	5.672	0.057	119.547	1.195	<ipython-input-4-5c83a5>:1(sort)
102/2	2.328	0.023	150.109	75.055	<string>:1(<listcomp>)

