4. Ausnahmen
und GIT

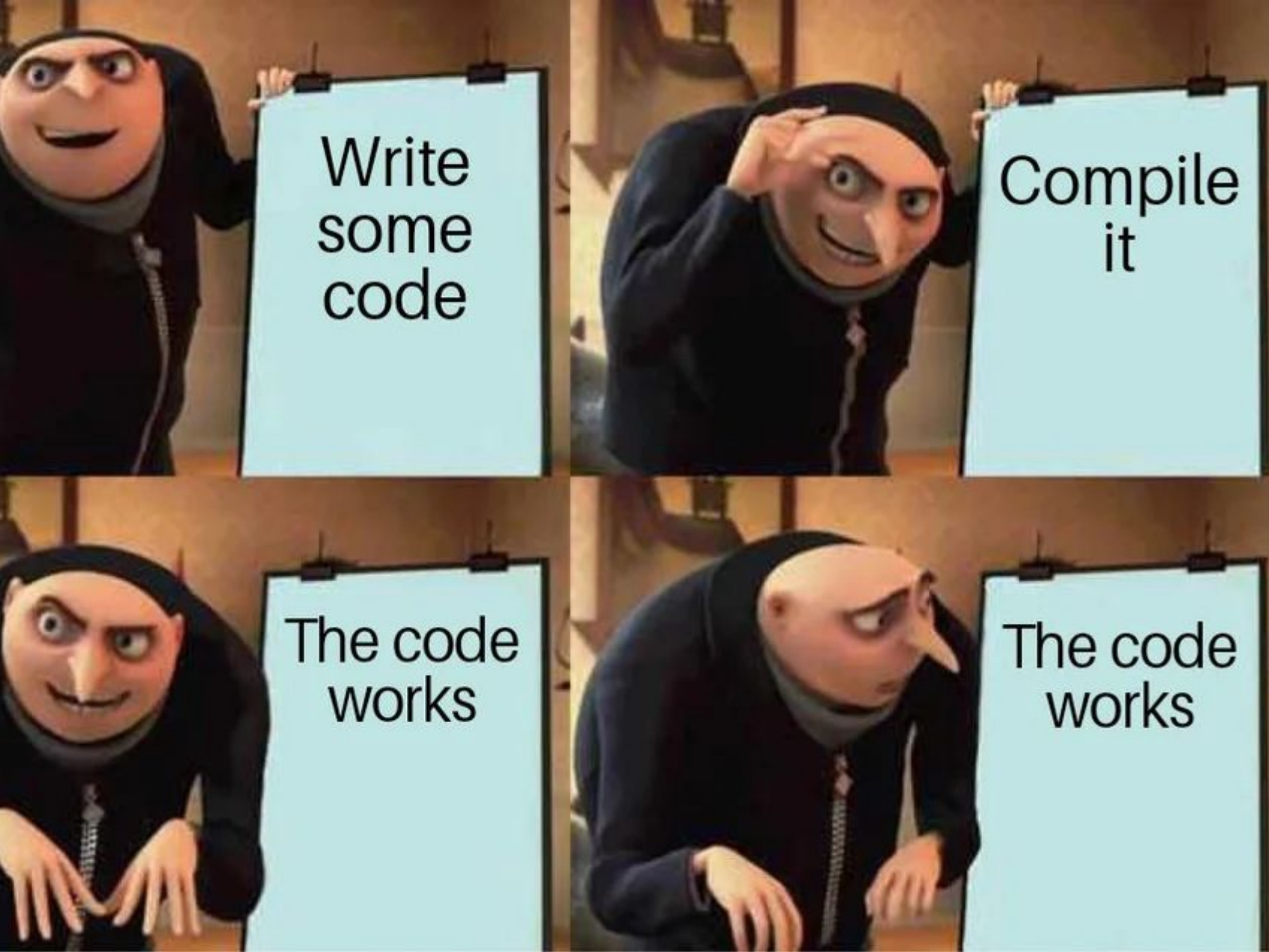
Inhalt

- Fehlerbehandlung
  - Spezifikationen

- Wie benutzt man GIT
  - und warum

Fehlerhafte Programme

- wenn wir Code schreiben…und…es einfach…funktioniert


Write some code
Compile it
The code works
The code works

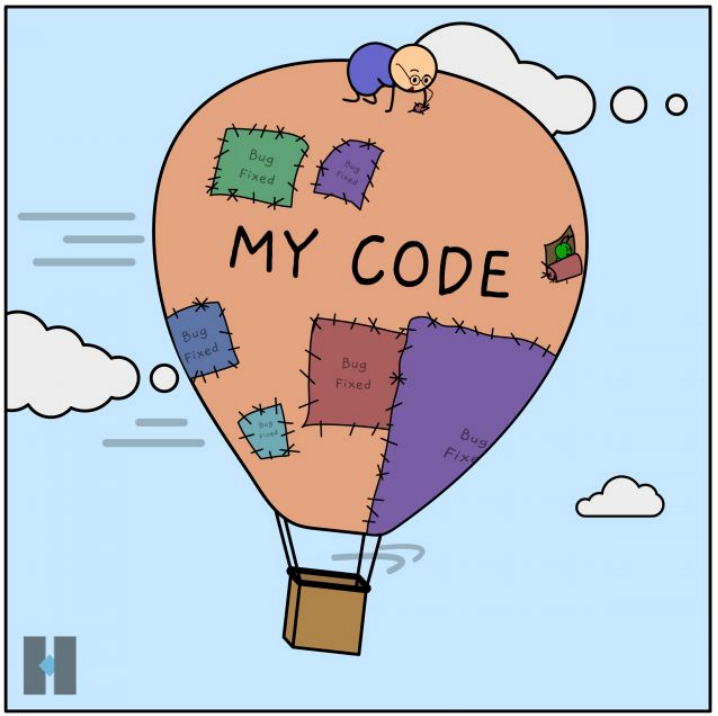
Fehlerhafte Programme

- wenn wir Code schreiben…und…es einfach…nicht funktioniert

Fehlerhafte Programme

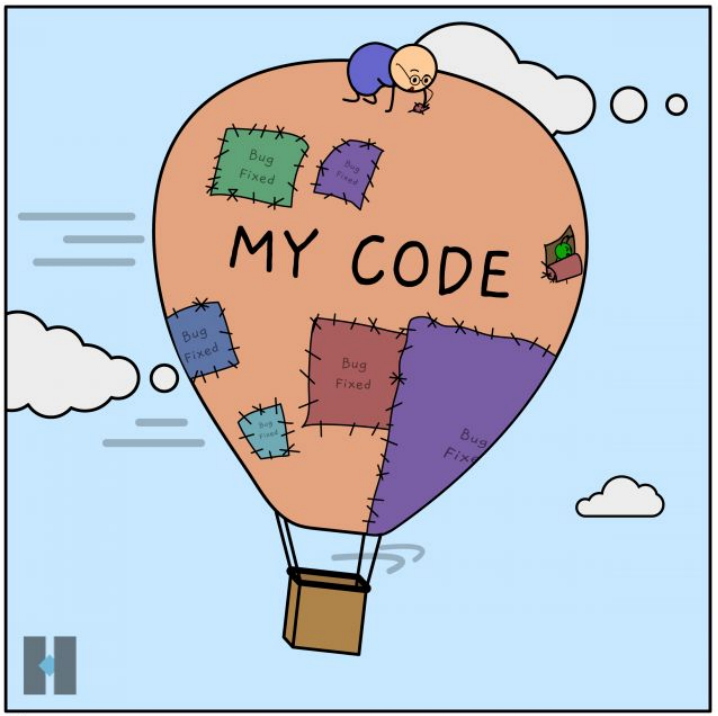
Fehlerhafte Programme

- Ein Programm kann aus vielen Gründen unerwünschtes Verhalten zeigen



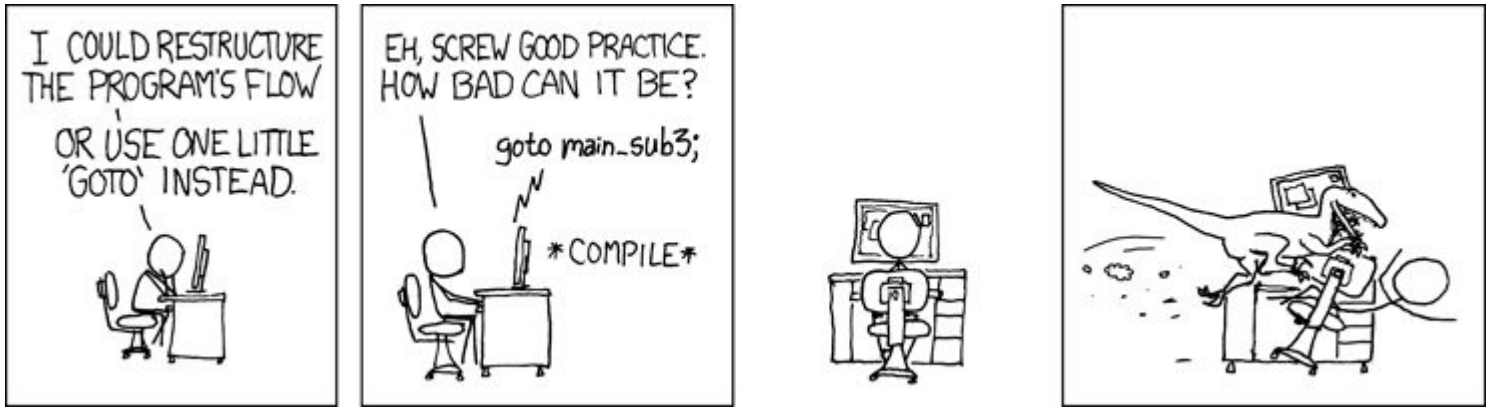
Fehlerhafte Programme

- Ein Programm kann aus vielen Gründen unerwünschtes Verhalten zeigen
- Bugs
- Fehler beim Entwurf
- Fehler bei der Programmierung des Entwurfs
  - Algorithmen falsch implementiert



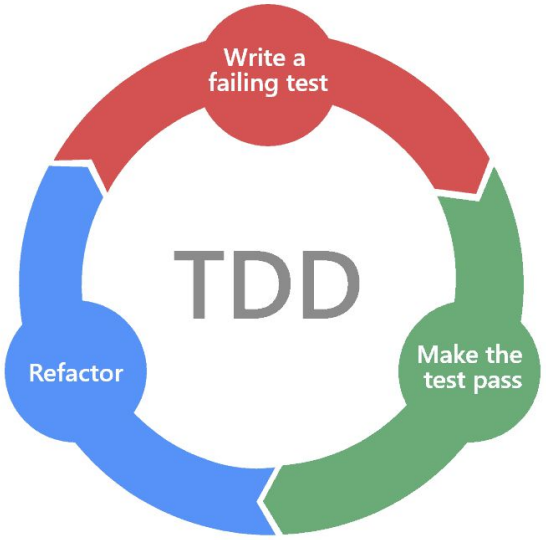
Fehlerhafte Programme

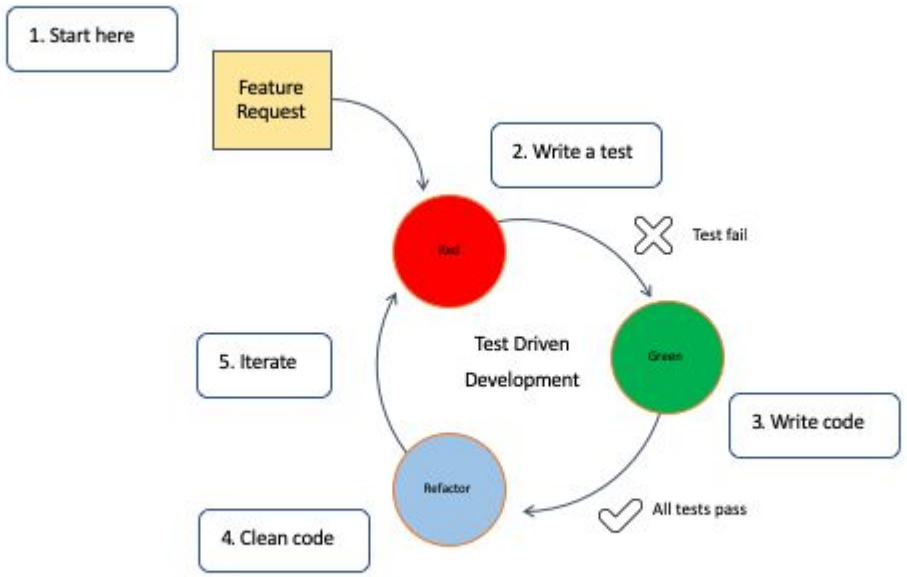
- das bedeutet dass der Entwickler etwas falsch bei der Implementierung gemacht hat
- Bug = das Programm funktioniert nicht wie man erwartet hätte



Test Driven Development

- beim Seminar erlebt
  - **beachte, wie ich Code schreibe!**
- Test-First-Ansatz
- wir stellen sicher, dass der Code mit Test abgedeckt ist
- wir können leichter Änderungen machen





Fehlerhafte Programme

- aber es gibt auch andere Situationen

```
def remove_element(l, pos):
    if pos >= 0:
        l.pop(pos)

l = []

remove_element(l, 0)
```

Fehlerhafte Programme

- aber es gibt auch andere Situationen

```
def remove_element(l, pos):
    if pos >= 0:
        l.pop(pos)

l = []

remove_element(l, 0)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop from empty list
>>>
```

Fehlerhafte Programme

- Ein Programm kann aus vielen Gründen unerwünschtes Verhalten zeigen

- außergewöhnlichen Situationen
  - Abbruch der Netzwerkverbindung
  - Dateien können nicht gefunden werden
  - fehlerhafte Benutzereingaben

- **eine so genannte Exception/Ausnahme**

Umgang mit außergewöhnlichen Situationen

- Ausnahmesituationen unterscheiden sich von Programmierfehlern darin, dass man sie nicht (zumindest prinzipiell) von vornherein ausschließen kann

- Immer möglich sind zum Beispiel:
  - unerwartete oder ungültige Eingaben
  - Ein- und Ausgabe-Fehler beim Zugriff auf Dateien oder Netzwerk

Ausnahmen

Eine **Ausnahme** (Exception) ist eine Ausnahmesituation, die sich während der Ausführung eines Programmes einstellt.

- Lässt man diese zu, so stürzt das Programm ab!
- Fängt man diese ab (Ausnahmebehandlung), läuft das Programm weiter!
- Die Auslösung einer Ausnahme bedeutet nicht automatisch, dass der Code einen Fehler enthält

Black Box/Code - kann eine Exception auslösen

Aufrufer/Client - kann die Exception abfangen

```
def black_magic():
        ….
```



```
def main ():
        black_magic()
```

Ausnahmen

Die meisten Programmiersprachen, die Ausnahmen unterstützen, verwenden eine gemeinsame Terminologie und Syntax

- Ausnahmen auslösen → raise
- Ausnahmen abfangen oder behandeln → try/except
- Verbreitung
- `try` / `raise` (`throw`) und `except` (`catch`) Keywords

Ausnahmebehandlung

- Ausnahmebehandlung (Exception handling)
  - der Prozess, bei dem Fehlerzustände in einem Programm systematisch behandelt werden

```
try:
   #Code der Ausnahmen auslösen kann
except <ErrorType>:
   #Code der die Situation beherrscht
```

- was hinter try kommt, wird ausgeführt, bis ein Fehler auftritt
- was hinter except kommt, wird nur ausgeführt, wenn im try-statement eine Exception der angegebenen Art aufgetreten ist
- Ja! Man muss die verschiedenen ErrorTypes wissen

Beispiel

wir haben den folgenden Code

```
def read_from_file(filename)
    """
    die Funktion liest einen String aus einer Datei ein
    """

    f = open(filename,"r")
    wort = f.read().strip()

    return wort
```

Beispiel

- wir haben den folgenden Code

```
def div(a,b):
    return a / b

print(div(1,2))
print(div(0,1))
print(div(1,0))
```

```
0.5
0.0
Traceback (most recent call last):
  File "main.py", line 6, in <module>
    print(div(1,0))
  File "main.py", line 2, in div
    return a / b
ZeroDivisionError: division by zero
```

- wir wollen diese Situation vermeiden
- wir konnen `if` verwenden
  - aber in dem Fall muss man extra Logik in der Funktion reinstecken

Beispiel

- Eleganter geht es mit Exceptions

```
def div(a,b):
    try:
        r = a / b
    except ZeroDivisionError:
        r = None
    return r

print(div(1,2))
print(div(0,1))
print(div(1,0))
```

```
0.5
0.0
None
```

Python Syntax

- Wenn man Ausnahmen abfangen will, muss der Code in einem `try-except` Block enthalten sein
- Ausnahmen werden anhand ihres Typs abgefangen
- Ein try-Block kann einen, mehrere oder alle Ausnahmetypen abfangen
- Das Erstellen bzw. Auslösen von Ausnahmen in unserem Code erfolgt mit dem Schlüsselwort `raise`
- Man kann zusätzliche Argumente (zB eine Fehlermeldung) für jede Ausnahme, die ausgelöst wird, bereitstellen

Ausnahmebehandlung

- Eine Ausnahme kann behandelt werden durch:
  - Die Funktion, bei der die Ausnahme ausgelöst wurde
  - Jede Funktion, die diese Funktion aufruft
  - der Python-runtime - dies wird zu einem Abbruch des Programmes führen

- der Satz ”unhandled exception has occurred in your application...” muss uns bekannt sein

- jetzt können wir verstehen, was dort passiert ist!

Types

- **SyntaxError** – es ist ein syntaktischer Fehler im Quelltext
- **IOError** – eine Datei existiert nicht, man darf nicht schreiben, die Platte ist voll
- **IndexError** – in einer Sequenz gibt es das angeforderte Element nicht
- **KeyError** – ein Mapping hat den angeforderten Schlüssel nicht
- **ValueError** – eine Operation kann mit diesem Wert nicht durchgeführt werden

None, NoneType, Pass

- `None` ist ein Objekt ohne Wert
- `NoneType` ist der Typ dieses Objektes
- None als Rückgabewert zeigt das
  - die Funktion retourniert nichts
  - zB eine Suchfunktion hat nichts gefunden
- `pass` ist eine Anweisung, die kein Ergebnis hat
  - nützlich, um Code zu strukturieren
  - kann verwendet werden, um zu zeigen
  - das code wird dort irgendwann geschrieben

```
def add(a,b):
    pass
```

None, NoneType, Pass

```
def useless_function():
    print("i am useless")

useless_function()

value = useless_function()
print(value)
print(type(value))  #<class ‘NoneType’>
```

Fehlerhafte Programme

- a wild function appears

```
def add(a, b):
    return a + b


print(add(1, 2))
print(add("a", "b"))
```

Fehlerhafte Programme

- man kann in Python Types nicht durchsetzen aber man kann Infos mit **Type Annotations** darüber geben

```
def add(a: int, b: int) -> int:
    return a + b


print(add(1, 2))
print(add("a", "b")) #warning, aber funktioniert
```

Ausnahmebehandlung

- Wie kann man überprüfen, ob ein Parameter einer Funktion von einem bestimmten Typ ist?

- mit type()/isinstance()

```
def add(a,b):
    if type(a) == int and typ(b) == int:
    # isinstance(a,int)
    ...
```

- mit Exceptions
  - TypeError/AttributeError

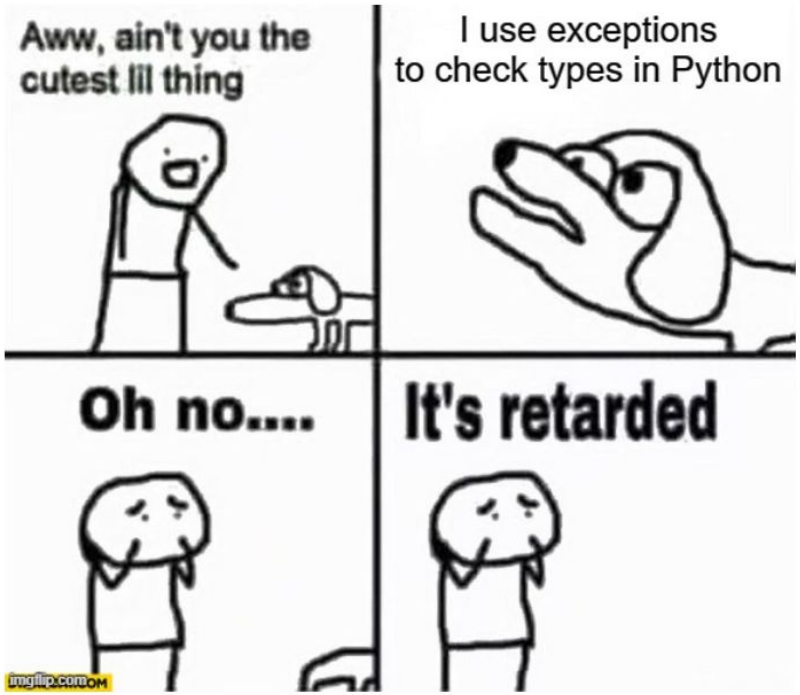
```
1  def my_max(arg):
2      try:
3          return max(arg)
4      except TypeError:
5          return 0
6
7
8  print(my_max([1,2,3]))
9  print(my_max(3))
10
```

Fehlerhafte Programme

- man könnte theoretisch in Python Types durchsetzen

```
def add(a,b):
    if type(a) == int and typ(b) == int:
        return a + b
    raise ValueError()

print(add(1, 2))
```



- Hinweis: kann man aber muss nicht
  - kann ein Indikator für schlechtes Design sein
  - die Spezifikation einer Funktion ist w

Type Annotations

- Type Annotations/Hints: int, list, str, float, dict, etc.
  - sind der richtige Weg in Python

- Read Me: https://peps.python.org/pep-0484/

```
def add(a: int, b: int) -> int:
    return a + b


print(add(1, 2))
print(add("a", "b")) #PyCharm warning: Expected type 'int'
```

Type Annotations

- Type Annotations/Hints: int, list, str, float, dict, etc.

```
from typing import List, NewType

Vector = NewType("Vector", List[float])

def scale(scalar: float, vector: Vector) -> Vector:
    new_vector = []
    for num in vector:
        new_vector.append(scalar*num)
    return new_vector

new_vector = scale(2.0, [1.0, -4.2, 2.0]) #PyChar Warning
print(new_vector)
new_vector2 = scale(2.0, Vector([1.0, -4.2, 2.0]))
print(new_vector2)
```

Intro

Git ist eine sehr verbreitete Versionsverwaltungssoftware

- was heißt das?
- wofür brauche ich das?
- wieso sollte mich das überhaupt interessieren?


```
> git checkout -b new_branch
> git merge new_branch
```

Typische Situation

Bearbeiten eines Projekts am Computer - unabhängig davon, ob es sich um Design, Programmcode oder Text handelt.

Ablauf

1. Datei anlegen
2. Datei speichern
3. Datei bearbeiten
4. Datei erneut speichern
5. Datei löschen

Track changes?

Ablauf

Ablauf, abstrakt

- Wiederkehrende Änderungen am Projekt
- Speichern einer Datei bringt uns zum nächsten Zustand

Zustand?

- T_i : aktueller Zustand (der Dateien) des Projekts
- T_{i-1}: vorherige Zustand, d.h. es wurde etwas geändert um in den jetzigen Zustand zu gelangen
- T_{i+1}: folgender Zustand, durch Änderung am aktuellen Zustand

Warum ist git wichtig?

Versionskontrolle

- ermöglicht Entwicklern, Änderungen am Quellcode zu verfolgen und zu verwalten
- erleichtert die Zusammenarbeit in Teams, da Entwickler gleichzeitig an verschiedenen Teilen des Codes arbeiten können, ohne sich in die Quere zu kommen
- behält den Verlauf aller Änderungen bei, was es einfach macht, zu früheren Versionen des Codes zurückzukehren, wenn Probleme auftreten

Kollaboration

- bietet Funktionen für das Zusammenarbeiten an Projekten
- mehrere Entwickler können gleichzeitig an einem Projekt arbeiten, indem sie eigene Versionen des Codes erstellen und später in den Hauptzweig integrieren
- das macht die Teamarbeit effizient und transparent

Beispiel

1. Erstellen eines neuen Git-Repositories in ‘git-test‘

```
git init git-test
cd git-test
```

2. Anlegen von ‘foo.txt‘

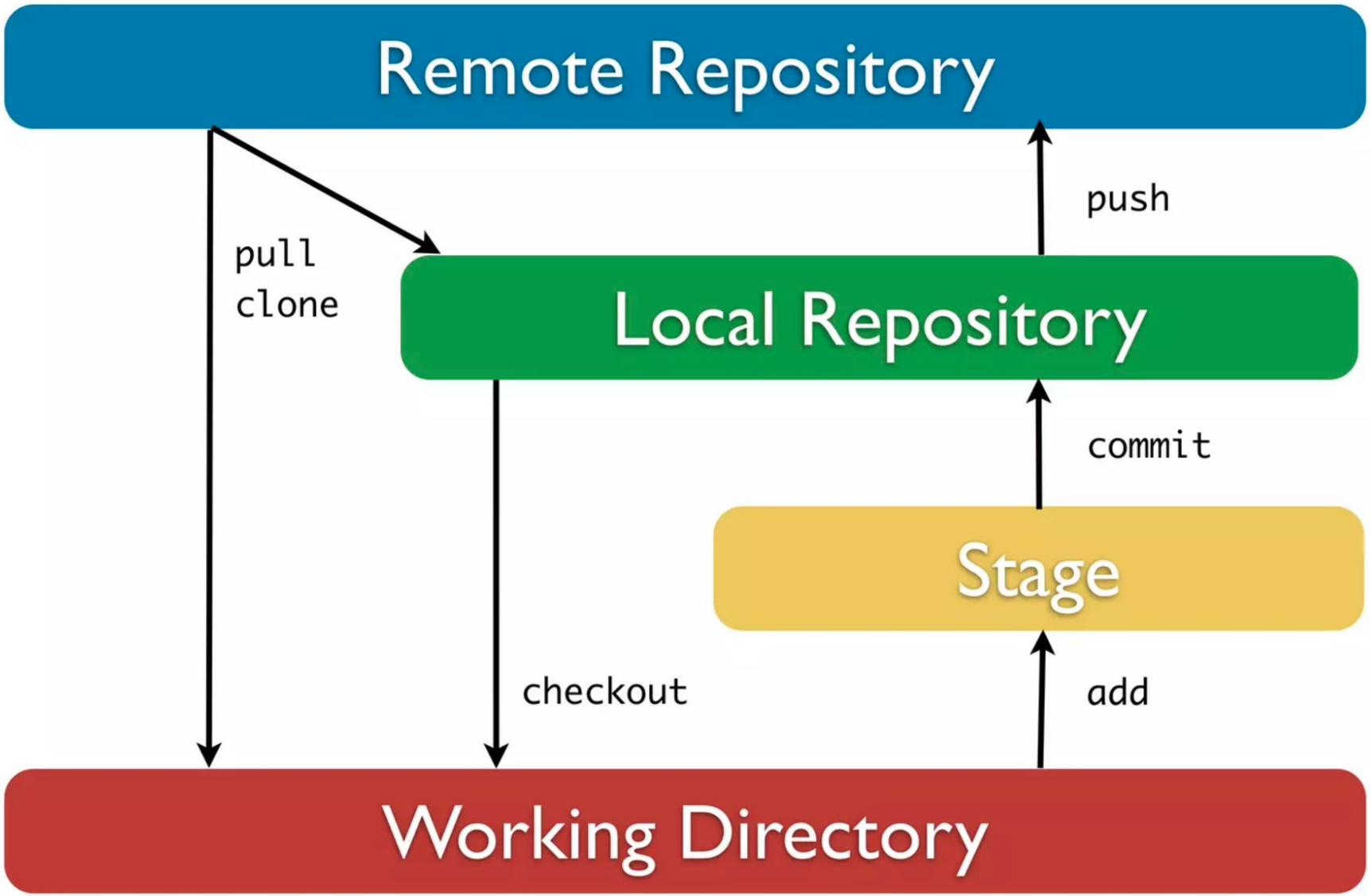
```
touch foo.txt
```

3. Commit verfassen, welcher ‘foo.txt‘ enthält

```
git add foo.txt
git commit -m "Created foo.txt"
```

```
#4. Commit hochschreiben
git push origin master
```

Ebenen


Remote Repository
pull
clone
push
Local Repository
commit
Stage
checkout
add
Working Directory

Repository Erstellen

Lokales Repository Erstellen

```
$ mkdir myrepo
$ cd myrepo
$ git init
```

```
Initialized empty Git repository in ../myrepo/.git
```

Arbeiten

```
$ echo "Hello World" > index.html
```

Status

Anzeigen aller geänderten und gelöschten Dateien

```
$ git status
```

Status

Anzeigen aller geänderten und gelöschten Dateien

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#     new file:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#     modified:   index.html
```

Staging

Alle oder mehrere bestimmte Dateien stagen

```
$ git add index.html
```

```
$ git add *.html
```

Alle Änderungen stagen (inkl. gelösten Dateien)

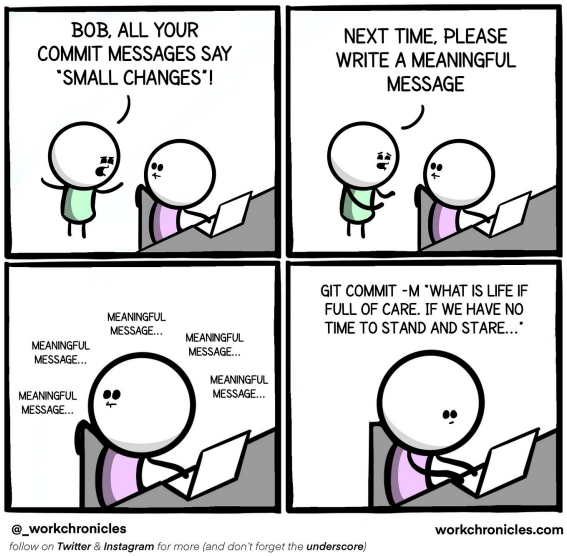
```
$ git add --all
```

Commit

Änderungen in Stage mit einer Nachricht ins lokale Repository committen

```
$ git commit -m "My first commit"
```

```
[master 7626937] changed
 1 files changed, 1 insertions(+), 1 deletions(-)
```



Stage & Commit

Alle änderungen stagen und sofort commiten

```
$ git commit -a -m "My Message"
```

```
[master 7626937] changed
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Letzte Commit-Message anpassen

```
$ git commit --amend
```

Remote Repository Clonen

git remote Repository in den Ordner “mydir” clonen

```
$ git clone git@192.168.123.212:testing.git mydir
```

```
Cloning into 'mydir'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
```

Commits Hochschreiben

Alle Commits zum Remote pushen

```
$ git push origin master
```

```
# Counting objects: 5, done.
# Writing objects: 100% (3/3), 272 bytes, done.
# Total 3 (delta 0), reused 0 (delta 0)
# To git@192.168.123.212:testing.git
#   edfec50..2fc284e  master -> master
```

Allgemeine Syntax

```
$ git push [remote-name] [remote-branch-name]
```

Exkurs: Git in action

1. Erstellen eines neuen Git-Repositories in git-tester
2. Anlegen von main.py
3. Commit verfassen, welcher main.py enthält
4. main.py abändern und neu committen
5. Neue Datei other.py erzeugen, main.py abändern
6. Commit anfertigen, welcher nur other.py enthält

Exkurs: Git in action

1.Erstellen eines neuen Git-Repositories in ‘git-tester‘
```
cd git-tester
git init git-tester
```

2.Anlegen von main.py
3.Commit verfassen
```
git add main.py
git commit -m "added main"
```

#4.Add main
```
git remote add origin <url>/git-tester
```

```
#5.Commit hochschreiben
git push origin master
```

#6. Änderung main1.py Anlegen other.py
#7.Commit verfassen
```
git add .
git commit -m "added main2 and change other"
```

```
#8.Commit hochschreiben
git push origin master
```

#9.Änderung main1.py other.py
#10.Commit verfassen
```
git add other.py
git commit -m "change only other"
```

```
#11.Commit hochschreiben
git push origin master
```