# VORLESUNG

## EINFÜHRUNG IN DIE PROGRAMMIERUNG FÜR COMPUTERLINGUISTEN

Leonie Weißweiler

# ANSAGE

- Ab dieser Woche in der Freitagsübung Wiederholungen
- Nächste Woche in der Vorlesung 20 Minuten Probeklausur
- Diese Folien werden auf meiner Website hochgeladen: http://www.cip.ifi.lmu.de/~weissweiler/

# GREEDY / NON-GREEDY

- Welchen Match eine Regex finden soll, ist nicht immer eindeutig
- Ein * oder + kann verschieden "weit" gehen
- `.*a` + `"Hey anna!"`
- `"Hey anna!"` oder `"Hey anna!"` ?
- Durch *Greedy* oder *Non-Greedy* kann der Unterschied spezifiziert werden!
- *Greedy* (englisch) **gierig**: wie gierig matcht der Regex?

# GREEDY / NON-GREEDY

- **Greedy**
  - Normales Verhalten
  - Matcht so weit wie **möglich**
  - `.*ein`
  - `Was für eine einsame Brücke?`
- **Non-Greedy**
  - Durch ein angehängtes Fragezeigen ausgelöst
  - Matcht nur so weit wie **nötig**
  - `.*?ein`
  - `Was für eine einsame Brücke?`

# AUSFÜHREN VON REGEXES

- Um eine Regex auszuführen, verwendet man in Python das Modul **`re`**
- Zuerst erzeugt man ein Regex-Objekt mit `regex = re.compile(r'M | [TN]|B')`
- Der Regex selbst muss in einem Raw-String stehen, damit man Backslashes direkt verwenden kann ohne dass sie von Python interpretiert werden

# AUSFÜHREN VON REGEXES

- Um herauszufinden **ob** eine Regex matcht verwendet man `re.search` und überprüft mit einem if
- ```
  import re
  text = 'We demand a shrubbery!'
  regex = re.compile(r'a \w+');

  m = re.search(regex, text)
  if m:
      print( 'Gefunden' )
  ```

  *>> Gefunden*


  ```

# AUSFÜHREN VON REGEXES

- Um herauszufinden **ob** eine Regex **am Anfang eines Strings** matcht verwendet man `re.match` und überprüft mit einem if
- ```
  import re
  text = 'We demand a shrubbery!'
  regex = re.compile(r'a \w+');

  m = re.match(regex, text)
  if m:
      print( 'Gefunden' )

  >>
  ```
  ```

# AUSFÜHREN VON REGEXES

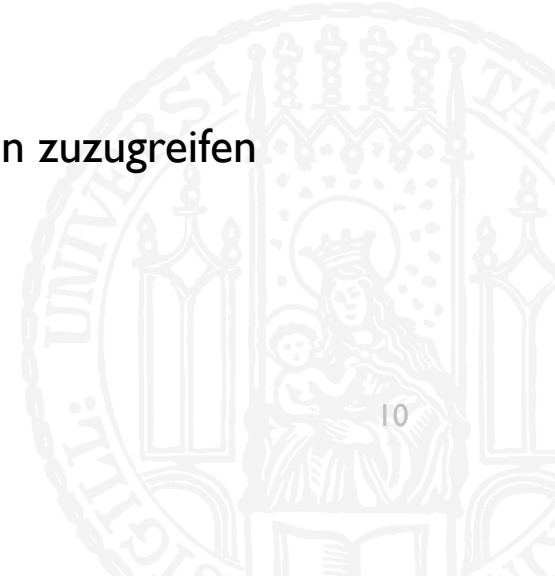
- Um **alle matches** eines Regexes zu finden verwendet man `re.findall` und erhält ein Array von Strings zurück
- ```
  import re
  text = 'We demand a shrubbery!'
  regex = re.compile(r'\w+');

  m = re.findall(regex, text)
  for match in m:
      print(match)
  ```

```
>> We
>> demand
>> a
>> shrubbery
```
```

# AUSFÜHREN VON REGEXES

- Um **alle matches** eines Regexes durch einen Text zu ersetzen verwendet man re.sub und erhält den neuen Text zurück
- ```
  import re
  text = 'We demand a shrubbery!'
  regex = re.compile(r'a \w+');

  neuer_text = re.sub(regex, "a fish", text)
  print (neuer_text)
  ```
  regex replacement string

*>> We demand a fish!*
  ```

# GROUPING

- Manchmal sind die einzelnen Teile einer Regex interessant.
- `\d \w+`
- `Das sind die 8 Ritter!`
- Man kann sie mit Groups einschließen und danach auf diese zugreifen
- `(\d) (\w+)`
- `Das sind die 8 Ritter!`
- Es ist sogar möglich später in der Regex auf vorherige Gruppen zuzugreifen
- `(\d) (\w+) und \1 (\w+)`
- `Das sind die 8 Ritter und 8 Zauberer!`



# AUSFÜHREN VON REGEXES

- Auf den Inhalt greift man mit `.group(n)` zu
- `.group(0)` gibt den ganzen match aus
- `.group(n)` gibt die n-te Klammer aus
- ```
  import re
  text = 'We demand a shrubbery!'
  regex = re.compile(r'a (\w+)');

  m = re.search(regex, text)
  if m:
      print( 'Gefunden: ' + m.group(0) )
      print( 'Es ist ein ' + m.group(1) )
  ```

```
>> Gefunden: a shrubbery
>> Es ist ein: shrubbery
```
```

# AUSFÜHREN VON REGEXES

- Bei Verwendung **einer** Gruppe gibt findall ein Array der Gruppeninhalte zurück
- ```
  import re
  text = 'We demand a shrubbery and a knight!'
  regex = re.compile(r‘a (\w+)');

  m = re.findall(regex, text)
  for match in m:
      print("Es ist ein " + match)
  ```

```
>> Es ist ein shrubbery
>> Es ist ein knight
```


```

# AUSFÜHREN VON REGEXES

- Bei Verwendung **mehrerer** Gruppem gibt findall ein Array von Tupeln zurück
- ```
  import re
  text = 'We demand a shrubbery and a knight!'
  regex = re.compile(r‘(a) (\w+)');

  m = re.findall(regex, text)
  for match in m:
      print(match)
  ```

```
>> ('a', 'shrubbery')
>> ('a', 'knight')
```


```

# FUNKTIONEN

```
a = 0
b = 15
c = 30

a_farenheit = (a * 9 / 5) + 32
print (a + " Grad Celsius sind " + a_farenheit + " Grad Farenheit!")
b_farenheit = (b * 9 / 5) + 32
print (b + " Grad Celsius sind " + b_farenheit + " Grad Farenheit!")
c_farenheit = (c * 9 / 5) + 32
print (c + " Grad Celsius sind " + c_farenheit + " Grad Farenheit!")
```

# FUNKTIONEN

- Es ist möglich mehrfach verwendete Programmabschnitte zu gruppieren
- Man nennt diese Gruppen Funktionen und kann sie danach beliebig oft wieder aufrufen
- Eine Funktion wird mit dem Keyword **`def`**, einem Namen und `():` eingeleitet
- Danach kann sie durch `name()` beliebig aufgerufen werden
- ```
  def hallo_sagen():
      print('Hallo')

  hallo_sagen()
  ```
- **Faustregel:** anstatt Copy+Paste eine Funktion schreiben
  ```

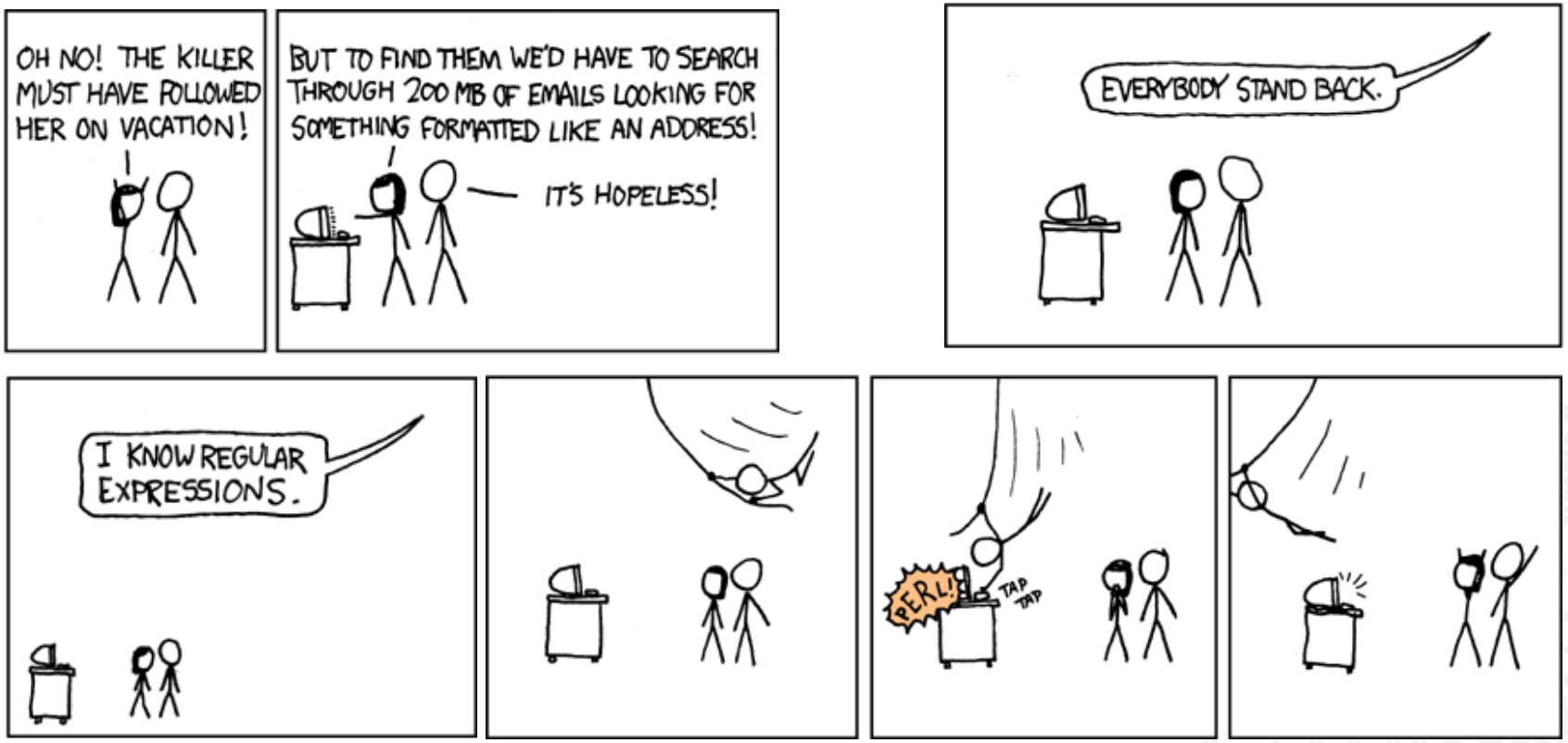
# FUNKTIONEN

- Mit dem Keyword **return** kann eine Funktion auch etwas zurückliefern
- ```
  def fünf_fakultät():
      return 5*4*3*2*1

  ergebnis = fünf_fakultät()
  ```
- ```
  def aktuelles_jahr():
      return 2017

  jahr = aktuelles_jahr()
  ```

# FUNKTIONEN

```
def c_to_f(celsius):
    farenheit = (celsius * 9 / 5) + 32
    print (celsius + " Grad Celsius sind " + farenheit + " Grad Farenheit!")


a = 0
b = 15
c = 30

c_to_f(a)
c_to_f(b)
c_to_f(c)
```

# FALLSTUDIE


OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP