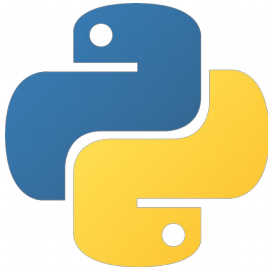
Python Strings

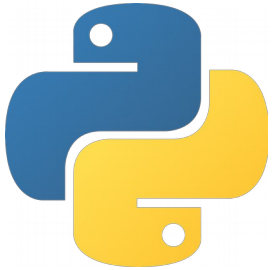


- Reguläre Ausdrücke – re
  - Extensions
    - Zusätzlich zu dieser mehr oder weniger standardisierten Syntax erlaubt Python die Verwendung sogenannter Extensions
    - Eine Extension ist folgendermaßen aufgebaut

      (?...)

    - Die drei Punkte werden durch eine Kennung der gewünschten Extension und weitere extensionspezifische Angaben ersetzt
    - Diese Syntax wurde gewählt, da eine öffnende Klammer, gefolgt von einem Fragezeichen, keine syntaktisch sinnvolle Bedeutung hat und demzufolge »frei« war

Python Strings

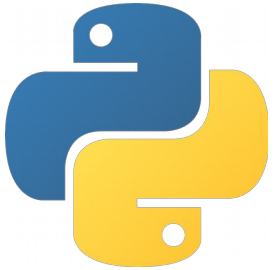


- Reguläre Ausdrücke – re
  - Extensions
    - Beachte
      - Eine Extension muss nicht zwingend eine neue Gruppe erzeugen, auch wenn die runden Klammern dies nahelegen

      (?aiLmsux)

    - Diese Extension erlaubt es, ein oder mehrere Flags für den gesamten regulären Ausdruck zu setzen
    - Dabei bezeichnet jedes der Zeichen »a«, »i«, »L«, »m«, »s«, »u« und »x« ein bestimmtes Flag
      → später mehr dazu ←

Python Strings

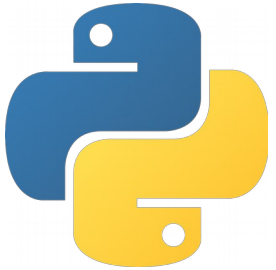


- Reguläre Ausdrücke – re
  - Extensions
    - Das Flag → i
      - macht den regulären Ausdruck beispielsweise case-insensitive

    r"(?i)P"

    Dieser Ausdruck passt sowohl auf "P" als auch auf "p".

Python Strings



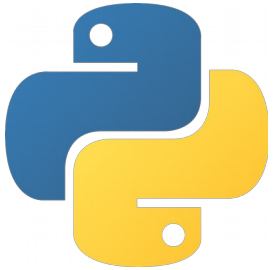
- Reguläre Ausdrücke – re
  - Extensions

    (?:...)

    - Diese Extension wird wie normale runde Klammern verwendet, erzeugt dabei aber keine Gruppe
    - Das heißt, auf einen durch diese Extension eingeklammerten Teilausdruck können Sie später nicht zugreifen
    - Mithilfe dieses Konstrukts lässt sich ein regulärer Ausdruck strukturieren, ohne dass dabei ein Overhead durch die Bildung von Gruppen entsteht

    - Äquivalent
      r"(?:abc|def)"

Python Strings



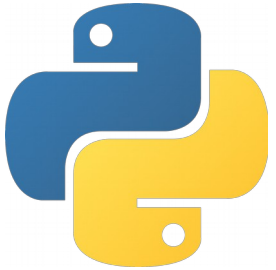
- Reguläre Ausdrücke – re
  - Extensions

    (?P<name>...)

    - Diese Extension erzeugt eine Gruppe mit dem angegebenen Namen
    - Das Besondere an einer solchen benannten Gruppe ist, dass sie nicht allein über ihren Index, sondern auch über ihren Namen referenziert werden kann
    - Namensreglen beachten

    r"(?P<hallowelt>abc|def)"

Python Strings



- Reguläre Ausdrücke – re
  - Extensions

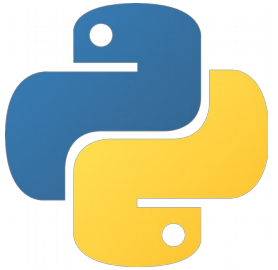
    (?P=name)

    - Diese Extension passt auf all das, auf das die bereits definierte Gruppe mit dem Namen name gepasst hat
    - Diese Extension erlaubt es also, eine benannte Gruppe zu referenzieren

    r"(?P<py>[Pp]ython) ist, wie (?P=py) sein sollte"

    - Dieser reguläre Ausdruck passt auf den String "Python ist, wie Python sein sollte"

Python Strings



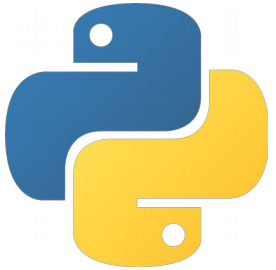
- Reguläre Ausdrücke – re
  - Extensions

    (?#...)

    - Diese Extension stellt einen Kommentar dar. Der Inhalt der Klammern wird schlicht ignoriert

    r"Py(?#lalala)thon"

Python Strings



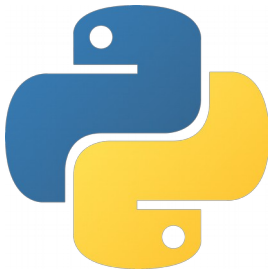
- Reguläre Ausdrücke – re
  - Extensions

    (?=...)

    - Diese Extension passt nur dann, wenn der reguläre Ausdruck ... als Nächstes passt
    - Diese Extension greift also vor, ohne in der Auswertung des Ausdrucks tatsächlich voranzuschreiten

    - Mit dem regulären Ausdruck r"\w+(?= Meier)" lässt sich beispielsweise nach den Vornamen aller im Text vorkommenden Meiers suchen
    - Die naheliegende Alternative r"\w+ Meier" würde nicht nur die Vornamen, sondern immer auch den Nachnamen Meier in das Ergebnis mit aufnehmen

Python Strings



- Reguläre Ausdrücke – re
  - Extensions

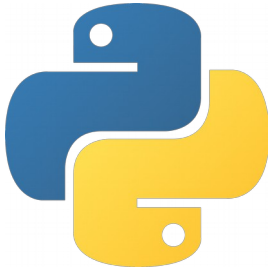
    (?!...)

    - Diese Extension passt nur dann, wenn der reguläre Ausdruck ... als Nächstes nicht passt. Diese Extension ist das Gegenstück zu der vorherigen

    (?<=...)

    - Diese Extension passt nur, wenn der reguläre Ausdruck ... zuvor gepasst hat
    - Diese Extension greift also auf bereits ausgewertete Teile des Strings zurück, ohne die Auswertung selbst zurückzuwerfen

Python Strings

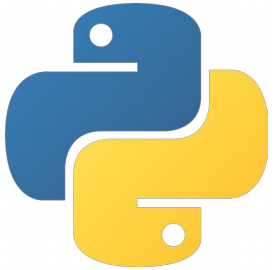


- Reguläre Ausdrücke – re
  - Extensions

    (?<!...)

    - Diese Extension passt nur, wenn der reguläre Ausdruck ... zuvor nicht gepasst hat
    - Diese Extension ist damit das Gegenstück zu der vorherigen

Python Strings

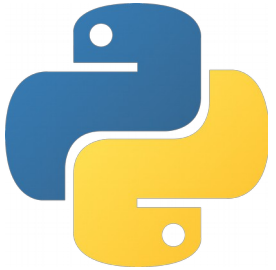


- Reguläre Ausdrücke – re
  - Extensions

    (?(id/name)yes-pattern|no-pattern)

    - Diese kompliziert anmutende Extension kann in einem regulären Ausdruck als eine Art Fallunterscheidung verwendet werden
    - Abhängig davon, ob eine Gruppe mit dem angegebenen Index bzw. dem angegebenen Namen auf einen Teilstring gepasst hat, wird entweder (im positiven Fall) auf das yes-pattern oder (im negativen Fall) auf das no-pattern getestet
    - Das no-pattern wird durch einen senkrechten Strich vom yes-pattern getrennt, kann aber auch weggelassen werden

Python Strings



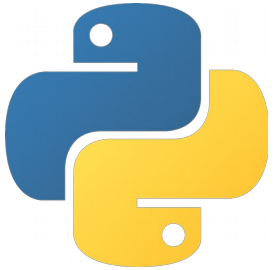
- Reguläre Ausdrücke – re
  - Extensions

    r"(?P<klammer>\()?Python(?(klammer)\))"

    - In diesem Ausdruck wird zunächst eine Gruppe namens klammer erstellt, die maximal einmal vorkommen darf und aus einer öffnenden, runden Klammer besteht
    - Danach folgt die Zeichenkette Python , und schlussendlich wird durch die Extension eine schließende Klammer gefordert, sofern zuvor eine öffnende aufgetreten ist, also sofern die Gruppe klammer zuvor gepasst hat

    Damit passt der reguläre Ausdruck auf die Strings "Python" und "(Python)", beispielsweise aber nicht auf "(Python"

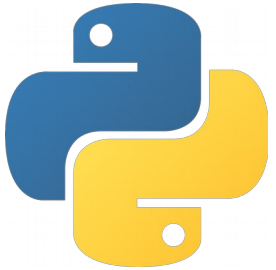
Python Strings



- Reguläre Ausdrücke – re
  - Verwendung des Moduls re

Funktion	Beschreibung
compile(pattern[,flags])	Kompiliert den regulären Ausdruck *pattern* zu einem Regular-Expression-Objekt.
escape(string)	Ersetzt problematische Zeichen in *string* durch ihre Escape-Sequenzen.
findall(pattern,string [,flags])	Sucht nach allen Teilen von *string*, die auf den regulären Ausdruck *pattern* passen.
finditer(pattern,string [,flags])	Wie *findall*, gibt das Ergebnis aber als Iterator über Match-Objekte und nicht als Liste von Strings zurück.

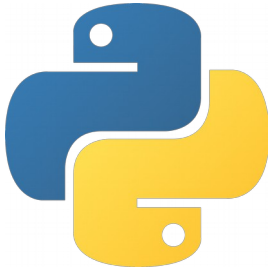
Python Strings



- Reguläre Ausdrücke – re
  - Verwendung des Moduls re

Funktion	Beschreibung
match(pattern,string [,flags])	Prüft, ob *string* auf den regulären Ausdruck *pattern* passt.
purge()	Löscht den internen Cache des Moduls.
search(pattern,string [,flags])	Sucht in *string* nach einem Teil, der auf den regulären Ausdruck *pattern* passt.
split(pattern,string [,maxsplit[,flags]])	Spaltet *string* an Stellen auf, die auf den regulären Ausdruck *pattern* passen.
sub(pattern,repl,string [,count[,flags]])	Ersetzt auf *pattern* passende Teilstrings von *string* durch *repl*.
subn(pattern,repl,string [,count[,flags]])	Wie *sub*, gibt aber zusätzlich die Anzahl der vorgenommenen Ersetzungen zurück.

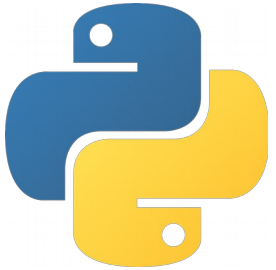
Python Strings



- Reguläre Ausdrücke – re
  - Verwendung des Moduls re
  - Einbinden → import re
  - Funktion → escape(string)

    >>> re.escape("Funktioniert das wirklich? ... (ja!)")
    'Funktioniert\\ das\\ wirklich\\?\\ \\.\\.\\.\\ \\(ja\\!\\)'

Python Strings



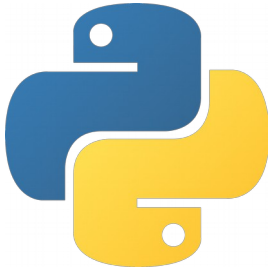
- Reguläre Ausdrücke – re
  - Verwendung des Moduls re
  - Funktion → findall(pattern, string, flags=0)

    ```
    >>> re.findall(r"P[Yy]thon", "Python oder PYthon und Python")
    ['Python', 'PYthon', 'Python']
    ```

  - Wenn pattern ein oder mehrere Gruppen enthält, werden diese anstelle der übereinstimmenden Teilstrings in die Ergebnisliste geschrieben

    ```
    >>> re.findall(r"P([Yy])thon", "Python oder PYthon und Python")
    ['y', 'Y', 'y']
    >>> re.findall(r"P([Yy])th(.)n", "Python oder PYthon und Python")
    [('y', 'o'), ('Y', 'o'), ('y', 'o')]
    ```

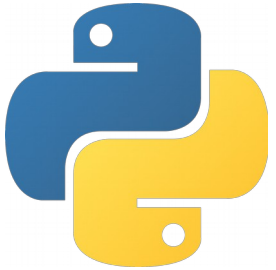
Python Strings



- Reguläre Ausdrücke – re
  - Verwendung des Moduls re
  - Funktion → match(pattern, string, flags=0)

```
>>> print(re.match(r"P[Yy]thon", "PYYthon"))
None
>>> re.match(r"P[Yy]thon", "PYthon")
<_sre.SRE_Match object at 0xb7bd7f00>
```

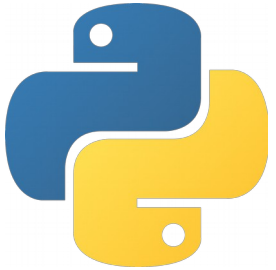
Python Strings



- Reguläre Ausdrücke – re
  - Verwendung des Moduls re
  - Funktion → search(pattern, string, flags=0)
  - Wenn kein Ergebnis gefunden wurde, gibt die Funktion None zurück

```
>>> re.search(r"P[Yy]thon", "Nimm doch Python")
<_sre.SRE_Match object at 0xb7bd7f00>
```

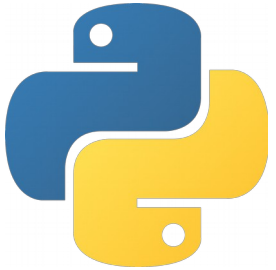
Python Strings



- Reguläre Ausdrücke – re
  - Verwendung des Moduls re
  - Funktion → split(pattern, string, maxsplit=0, flags=0)
  - Der String string wird nach Übereinstimmungen mit dem regulären Ausdruck pattern durchsucht
  - Alle passenden Teilstrings werden als Trennzeichen angesehen, und die dazwischenliegenden Teile werden als Liste von Strings zurückgegeben

```
>>> re.split(r"\s", "Python Python Python")
['Python', 'Python', 'Python']
```

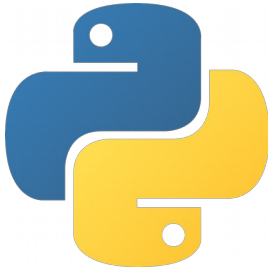
Python Strings



- Reguläre Ausdrücke – re
  - Verwendung des Moduls re
  - Funktion → split(pattern, string, maxsplit=0, flags=0)
  - Eventuell vorkommende Gruppen innerhalb des regulären Ausdrucks werden ebenfalls als Elemente dieser Liste zurückgegeben

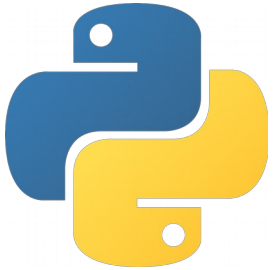
    >>> re.split(r"(,)\s", "Python, Python, Python")
    ['Python', ',', 'Python', ',', 'Python']
  - In diesem regulären Ausdruck werden alle von einem Whitespace gefolgten Kommata als Trennzeichen behandelt

Python Strings



- Reguläre Ausdrücke – re
  - Verwendung des Moduls re
  - Funktion → split(pattern, string, maxsplit=0, flags=0)
  - Wenn der Parameter maxsplit angegeben wurde und ungleich 0 ist, wird der String maximal maxsplit-mal unterteilt
  - Der Reststring wird als letztes Element der Liste zurückgegeben

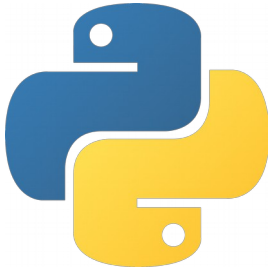
Python Strings



- Reguläre Ausdrücke – re
  - Verwendung des Moduls re
  - Funktion → sub(pattern, repl, string, count=0, flags=0)
  - Die Funktion sub sucht im String string nach nicht überlappenden Übereinstimmungen mit dem regulären Ausdruck pattern
  - Es wird eine Kopie des Strings string zurückgegeben, in dem alle passenden Teilstrings durch den String repl ersetzt wurden

    >>> re.sub(r"[Jj]a[Vv]a","Python", "Java oder java und jaVa")
    'Python oder Python und Python'

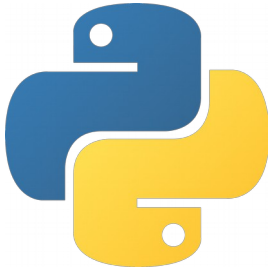
Python Strings



- Reguläre Ausdrücke – re
  - Verwendung des Moduls re
  - Funktion → subn(pattern, repl, string[, count])
  - Die Funktion subn funktioniert ähnlich wie sub, mit dem Unterschied, dass ein Tupel zurückgegeben wird, in dem zum einen der neue String und zum anderen die Anzahl der vorgenommenen Ersetzungen stehen

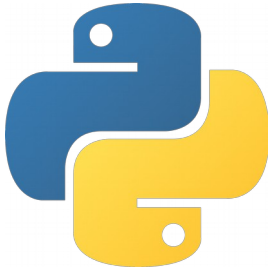
    >>> re.subn(r"([Jj]ava)","Python statt \g<1>", "Nimm doch Java")
    ('Nimm doch Python statt Java', 1)

Python Strings



- Reguläre Ausdrücke – re
  - Verwendung des Moduls re
  - Funktion → compile(pattern[, flags])
  - Diese Funktion kompiliert den regulären Ausdruck pattern zu einem Regular-Expression-Objekt
  - Bei mehreren Operationen auf demselben regulären Ausdruck lohnt es sich, diesen zu kompilieren, da diese Operationen dann schneller durchgeführt werden können

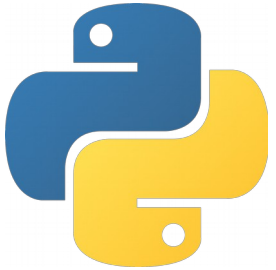
Python Strings



- Reguläre Ausdrücke – re
  - Verwendung des Moduls re
  - Funktion → compile(pattern[, flags])
  - Um die Auswertung des Ausdrucks zu beeinflussen, können Sie ein oder mehrere Flags angeben
  - Wenn es sich um mehrere handelt, müssen Sie sie durch das bitweise ODER | trennen

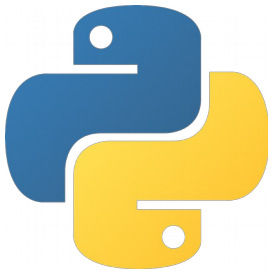
```
>>> c1 = re.compile(r"P[yY]thon")
>>> c2 = re.compile(r"P[y]thon", re.I)
>>> c3 = re.compile(r"P[y]thon", re.I | re.S)
```

Python Strings



- Reguläre Ausdrücke – re
  - Flags
    - Das sind bestimmte Einstellungen, die die Auswertung eines regulären Ausdrucks beeinflussen
    - Flags können Sie entweder im Ausdruck selbst durch eine Extension oder als Parameter einer der im Modul re verfügbaren Funktionen angeben
    - Sie beeinflussen nur den Ausdruck, der aktuell verarbeitet wird, und verbleiben nicht nachhaltig im System
    - Jedes Flag ist als Konstante im Modul re enthalten und kann über eine Lang- oder eine Kurzversion seines Namens angesprochen werden

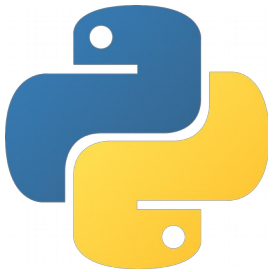
Python Strings



- Reguläre Ausdrücke – re
  - Flags

Alias	Name	Bedeutung
re.A	*re.ASCII*	Beschränkt die Zeichenklassen \w, \W, \b, \B, \s und \S auf den ASCII-Zeichensatz.
re.I	*re.IGNORECASE*	Macht die Auswertung des regulären Ausdrucks *case insensitive*, das heißt, dass die Zeichengruppe [A-Z] sowohl auf Groß- als auch auf Kleinbuchstaben passen würde.
re.L	*re.LOCALE*	Gibt an, dass bestimmte vordefinierte Zeichenklassen von der aktuellen Lokalisierung abhängig gemacht werden sollen. Das betrifft die Gruppen \w, \W, \b, \B, \s und \S.
re.M	*re.MULTILINE*	Wenn dieses Flag gesetzt wurde, passt ^ sowohl zu Beginn des Strings als auch nach jedem Newline-Zeichen und $ vor jedem Newline-Zeichen. Normalerweise passen ^ und $ nur am Anfang bzw. am Ende des Strings.

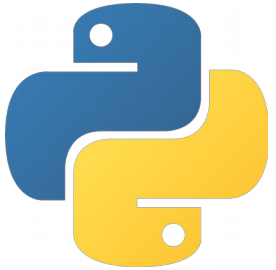
Python Strings



- Reguläre Ausdrücke – re
  - Flags

Alias	Name	Bedeutung
re.S	*re.DOTALL*	Wenn dieses Flag gesetzt wurde, passt das Sonderzeichen » . « tatsächlich auf jedes Zeichen. Normalerweise passt der Punkt auf jedes Zeichen außer auf das Newline-Zeichen \n.
re.U	*re.UNICODE*	Wenn dieses Flag gesetzt wurde, passen sich die vordefinierten Zeichenklassen dem Unicode-Standard an. Das heißt, dass dann auch Nicht-ASCII-Zeichen als Buchstabe oder Ziffer eingestuft werden. Dieses Flag ist seit Python 3.0 standardmäßig gesetzt.
re.X	*re.VERBOSE*	Das Setzen dieses Flags erlaubt es Ihnen, einen regulären Ausdruck zu formatieren. Wenn es gesetzt wurde, werden Whitespace-Zeichen wie Leerzeichen, Tabulatoren oder Newline-Zeichen ignoriert, solange sie nicht durch einen Backslash eingeleitet werden. Zudem leitet ein #-Zeichen einen Kommentar ein. Das heißt, alles hinter diesem Zeichen bis zu einem Newline-Zeichen wird ignoriert.

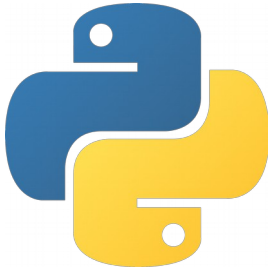
Python Strings



- Reguläre Ausdrücke – re
  - Das Match-Objekt
  - Erläuterungen im folgenden Kontext
    >>> import re
    >>> m = re.match(r"(P[Yy])(th.n)", "Python")

    m.expand(template)
  - Die Methode expand erlaubt es, den String template mit Informationen zu füllen, die aus der Matching- bzw. Searching-Operation stammen
  - So können über \g<index> und \g<name> die Teilstrings eingefügt werden, die auf die jeweiligen Gruppen gepasst haben

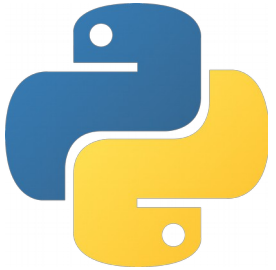
Python Strings



- Reguläre Ausdrücke – re
  - Das Match-Objekt
  - Erläuterungen im folgenden Kontext
    >>> import re
    >>> m = re.match(r"(P[Yy])(th.n)", "Python")

    m.group([group1, ...])
  - Die Methode group erlaubt einen komfortablen Zugriff auf die Teilstrings, die auf die verschiedenen Gruppen des regulären Ausdrucks gepasst haben
  - Wenn nur ein Argument übergeben wurde, ist der Rückgabewert ein String, ansonsten ein Tupel von Strings

Python Strings



- Reguläre Ausdrücke – re
  - Das Match-Objekt

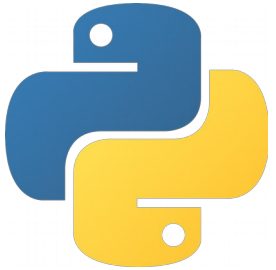
  - Erläuterungen im folgenden Kontext
    ```
    >>> import re
    >>> m = re.match(r"(P[Yy])(th.n)", "Python")

    m.group([group1, ...])
    ```

  - Wenn eine Gruppe auf keinen Teilstring gepasst hat, wird für diese None zurückgegeben

    ```
    >>> m.group(1)
    'Py'
    >>> m.group(1, 2)
    ('Py', 'thon')
    ```

Python Strings



- Reguläre Ausdrücke – re
  - Das Match-Objekt

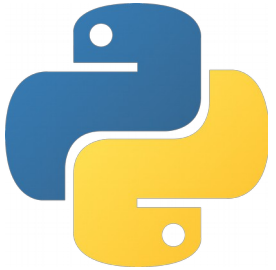
  - Erläuterungen im folgenden Kontext
    ```
    >>> import re
    >>> m = re.match(r"(P[Yy])(th.n)", "Python")

    m.group([group1, ...])
    ```

  - Ein Index von 0 gibt den vollständigen passenden String zurück.

    ```
    >>> m.group(0)
    'Python'
    ```

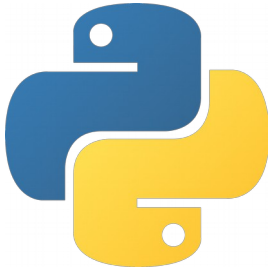
Python Strings



- Reguläre Ausdrücke – re
  - Das Match-Objekt
  - Erläuterungen im folgenden Kontext
    >>> import re
    >>> m = re.match(r"(P[Yy])(th.n)", "Python")

    m.groups([default])
  - Diese Methode gibt ein Tupel zurück, das alle Teilstrings enthält, die auf eine der im regulären Ausdruck enthaltenen Gruppen gepasst haben
  - Der optionale Parameter default erlaubt es, den Wert festzulegen, der in das Tupel geschrieben wird, wenn auf eine Gruppe kein Teilstring gepasst hat

Python Strings



- Reguläre Ausdrücke – re
  - Das Match-Objekt

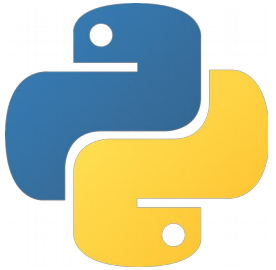
  - Erläuterungen im folgenden Kontext
    ```
    >>> import re
    >>> m = re.match(r"(P[Yy])(th.n)", "Python")

    m.groups([default])
    ```

  - Der Parameter ist mit None vorbelegt

    ```
    >>> m.groups()
    ('Py', 'thon')
    ```

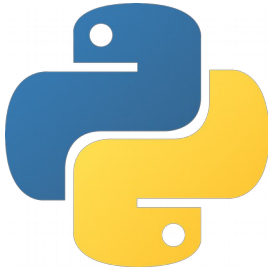
Python Strings



- Reguläre Ausdrücke – re
  - Das Match-Objekt
  - Erläuterungen im folgenden Kontext
    >>> import re
    >>> m = re.match(r"(P[Yy])(th.n)", "Python")

    m.groupdict([default])
  - Diese Methode gibt ein Dictionary zurück, das die Namen aller benannten Gruppen als Schlüssel und die jeweils passenden Teilstrings als Werte enthält
  - Der Parameter default hat die gleiche Bedeutung wie bei der Methode groups

Python Strings



- Reguläre Ausdrücke – re
  - Das Match-Objekt

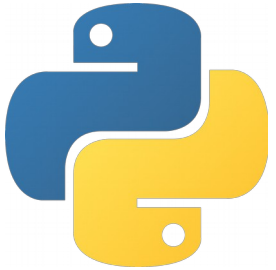
  - Erläuterungen im folgenden Kontext

```
>>> import re
>>> m = re.match(r"(P[Yy])(th.n)", "Python")

m.groupdict([default])

>>> c2 = re.compile(r"(?P<gruppe>P[Yy])(th.n)")
>>> m2 = c2.match("Python")
>>> m2.groupdict()
{'gruppe': 'Py'}
```

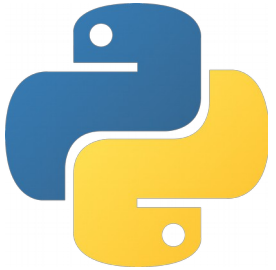
Python Strings



- Reguläre Ausdrücke – re
  - Das Match-Objekt
  - Erläuterungen im folgenden Kontext
    >>> import re
    >>> m = re.match(r"(P[Yy])(th.n)", "Python")

    m.start([group]), end([group])
  - Die Methode start gibt den Start- bzw. Endindex des Teilstrings zurück, der auf die Gruppe group gepasst hat
  - Der optionale Parameter group ist mit 0 vorbelegt

Python Strings

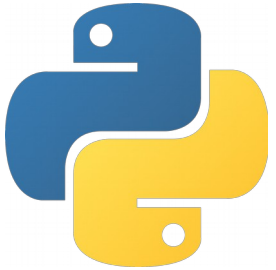


- Reguläre Ausdrücke – re
  - Das Match-Objekt
  - Erläuterungen im folgenden Kontext
    >>> import re
    >>> m = re.match(r"(P[Yy])(th.n)", "Python")

    m.start([group]), end([group])

    >>>m.start(2)
    2
    >>>m.end(2)
    6
  - Wenn eine Gruppe mehrfach gepasst hat, zählt der Teilstring, auf den die Gruppe letztmalig passte

Python Strings



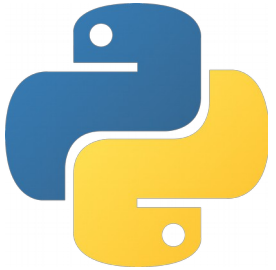
- Reguläre Ausdrücke – re
  - Aufgabe – Searching
    - Alle Links aus einer beliebigen HTML-Datei mitsamt Beschreibung herauszulesen

      Beschreibung

    - Ausgabe

      print("Name: {0}, Link: {1}".format(m.group(2), m.group(1)))

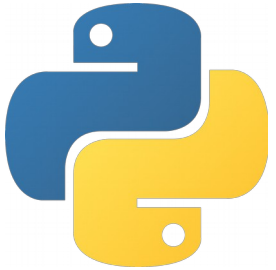
Python Strings



- Reguläre Ausdrücke – re
  - Aufgabe – Matching
    - Aus einer Art elektronischer Visitenkarte alle relevanten Informationen auslesen und maschinenlesbar aufbereiten

      Name: Max Mustermann
      Addr: Musterstr 123
      12345 Musterhausen
      Tel: +49 1234 56789

Python Strings



- Reguläre Ausdrücke – re
  - Aufgabe – Matching
    - Das Programm soll nun diese Textdatei einlesen, die enthaltenen Informationen extrahieren und zu einem solchen Dictionary aufbereiten

      {
      'Tel': ('+49', '1234', '56789'),
      'Name': ('Max', 'Mustermann'),
      'Addr': ('Musterstr', '123', '12345', 'Musterhausen')
      }

    - In der Textdatei soll dabei immer nur ein Datensatz stehen

    - Ausgabe → {'Name': ('Max', 'Mustermann')}