Empfohlene Unterrichtsinhalte als Vorbereitung für dieses Thema: 02-01/2/3/4 (Variablen, Strings, Listen, Integers und Floats), 03-03 (Schleifen)

Einführung in die computerlinguistische Programmierung mit Python 📖

10-01: String Formatting 📦

Wie ihr wisst, können die Datentypen, mit denen wir zu tun haben, mit der Methode `str()` explizit in Strings umgewandelt werden. So können wir Strings und Nicht-Strings mit dem `+` -Operator zusammenfügen und schließlich als einen konkatenierten String ausgeben:

In [1]:
```
frequencies = {"und": 1002, "oder": 876, "nicht": 311}

for word in frequencies:
    output = "Das Wort '" + word + "' kommt " + str(frequencies[word]) + "-
    print(output)
```

Es gibt einen sehr viel bequemeren Mechanismus, um Nicht-Strings in Strings einzubauen, und zwar die Methode `format()`. `format()` ist eine Stringmethode und nimmt beliebig viele Argumente an. Die Methode wird folgendermaßen verwendet:

In [1]:
```
frequencies = {"und": 1002, "oder": 876, "nicht": 311}

for word in frequencies:
    output = "Das Wort '{}' kommt {}-mal vor.".format(word, frequencies[wor
    print(output)
```

Die Methode `format()` operiert auf einem String in dem Platzhalter `{}` enthalten sind und nimmt Werte entgegen, die dann an Stelle der Platzhalter in den String eingesetzt werden. Dabei erfolgt die Typenumwandlung der jeweiligen Werte automatisch.

Somit können wir auf das Zusammensetzen von Strings und Nicht-Strings mit dem `+` -Operator verzichten.

Grundsätzlich werden die `format()` -Argumente in ihrer angegebenen Reihenfolge in den String eingesetzt. Sie lassen sich aber auch an Hand ihrer Position im Format-String referenzieren.

In [1]:
```
sent1 = "{} loves {}.".format("Mary", "John")
sent2 = "{1} loves {0}.".format("Mary", "John")
sent3 = "{0} loves {0}.".format("Mary", "John")

print("{0}{3}{1}{3}{2}".format(sent1, sent2, sent3, "\n---\n"))
```

Die nicht benötigten Argumente, die an `format()` übergeben werden, werden ignoriert.

Formatierung und komplexe Datentypen

Wir müssen nicht immer einzelne Argumente für die `format()` -Methode bereitstellen. Wir können auch Elemente aus strukturiertenn Datentypen (Listen, Dictionaries, Tupel) verarbeiten, die wir beispielsweise als Rückgabewert von anderen Funktionen erhalten, um

diese in übersichtlicherer oder besser lesbarer Form auszugeben.

Listen

Elemente aus Listen in Format-Strings eingesetzt werden in dem man die Liste als Argument ein **benanntes Argument** übergibt. Über den Argumentnamen können wir dann auf per Indexing auf die Listenelemente zugreifen.

In [1]:
```
constituents = ["Tom", "respects", "Avery"]
print("{data[0]} {data[1]} {data[2]}.".format(data=constituents))
```

Hier erhält `data` den Wert der vorher definierten Liste und wir können im Format-String Elemente der Liste mit Bezug auf den Namen des Arguments einsetzen lassen. Die gescheiften Klammern bewirken hier, dass `data` in diesem Fall **nicht** nur als einfacher String interpretiert wird, der in den formatierten String übernommen wird.

Wenn eine Liste als unbenanntes Argument übergeben wird, können wir auf deren Elemente mit Hilfe ihres Index entsprechend der Position in der Argumentliste zugreifen.

In [1]:
```
constituents = ["Tom", "respects", "Avery"]
print("{0[0]} {0[1]} {0[2]}.".format(constituents))
```

Die `0` referenziert die Liste `constituents`, die als nulltes Argument an `format()` übergeben wurde.

Man kann Elemente aus Sequenzen auch mit weniger Schreibarbeit als Argumente an die `format()` -Methode übergeben. Dabei hilft der **Unpacking-Operator** `*` : Die Elemente der Liste werden als einzelne unbenannte Argumente an die `format()` -Methode übergeben.

In [1]:
```
constituents = ["Tom", "respects", "Avery"]
print("{} {} {}.".format(*constituents))
```

Der Code mit dem `*` oben entspricht dem folgenden Code, in dem alle Argumente einzeln durch Komma getrennt übergeben werden:

In [1]:
```
print("{} {} {}.".format("Tom", "respects", "Avery"))
```

Mit Hilfe von Indexangaben können bestimmte Elemente aus einer längeren Liste ausgewählt werden um in den String eingefügt zu werden.

In [1]:
```
constituents = ["Lucky", "Kim", "carefully", "gave", "attentive", "Ashley",
print("{1} {3} {5} {6}.".format(*constituents))
```

Dictionaries

Wie Listen können wir auch Dictionaries an die `format()` -Methode übergeben. Auch hier haben wir verschieden Möglichkeiten. Wir können ein Dictionary als benanntes Argument übergeben und dann mit Hilfe der Schlüssel auf einezelne Werte zugreifen.

In [1]:
```
name_dict = {"title": "Dr.", "first" : "Angela Dorothea", "last" : "Merkel"
print("Erfasster Name: {data[title]} {data[first]} {data[last]}".format(dat
```

Um auch hier lässt sich mit Hilfe eines weiteren Unpacking Operators `**` speziell für Dictionaries Schreibarbeit sparen: Die Schlüssel-Wert-Paare des Dictionaries werden so behandelt als wären sie als einzelne benannte Argumente an `format[]` übergeben worden.

In [1]:
```
name_dict = {"title": "Dr.", "first" : "Angela Dorothea", "last" : "Merkel"
print("Erfasster Name: {title} {first} {last}".format(**name_dict))
```

Der `**` führt so dazu, dass der Code oben dem folgenden Code mit mehreren benannten Argumenten entspricht:

In [1]:
```
print("Erfasster Name: {title} {first} {last}".format(title = "Dr.", first
```

Wenn wir im Zusammenhang mit Dictionaries den einfachen `*`-Operator einsetzen, können wir Format-String nicht mehr die Schlüssel benutzen um auf die Werte referenzieren. Stattdessen werden **nur die Schlüssel** selbst als unbenannte Argumente bereitgestellt.

In [1]:
```
# name_dict = {"title": "Dr.", "first" : "Angela Dorothea", "last" : "Merke
# print("Erfasster Name: {title} {first} {last}".format(*name_dict))

print("Hmmmm...")

name_dict = {"title": "Dr.", "first" : "Angela Dorothea", "last" : "Merkel"
print("Erfasster Name: {2}, {0} {1}".format(*name_dict))
```

Mehr Informationen zu den Operatoren: Unpacking Operators in Python

Achtung! Zu wenige Argumente machen Probleme

Wenn wir weniger Argumente bereitstellen als Platzhalter in unserem Format-String vorhanden sind, wirft der Interpreter einen Fehler.

In [1]:
```
print("{0}! {1} {2}".format("Achtung", "jetzt"))
```

Dieser Fall muss abgefangen werden, wenn wir größere Datenmengen verarbeiten und `format()` mit automatisch erstellten Sequenzen gefüttert wird.

Formatierungs-Syntax

Padding und Alignment

Über eine spezielle Syntax für Format-String-Platzhalter lassen sich auch die Ausrichtung der Ausdrücke und der umgebende Whitespace beeinflussen. Grundsätzlich wird genau soviel Platz für einen einzusetzenden Ausdruck benutzt wie nötig ist. Wir können aber können aber auch eine bestimmte Länge für einen Platzhalter reservieren.

In [1]:
```
print("-->{:40}<--".format("Hier ist Platz für 40 Zeichen"))
```

Der Doppelpunkt innerhalb der geschweiften Klammern zeigt an, dass an dieser Stelle die Formatierungsinformationen folgen.

Wenn der einzusetzende String mehr als den vorgegebenen Raum benötigt wird die Vorgabe ignoriert.

In [1]:
```
langer_text = "Ich hab', ich hab', ich hab', ich hab', ich hab', zu wenig
print(len(langer_text))
print("-->{:40}<--".format(langer_text))
```

Per Default wird der einzusetzende Ausdruck innerhalb seines Slots links ausgerichtet. Wir können explizit andere Ausrichtungen erzwingen:

In [1]:
```
print("-->{:<25}<--".format("links"))
print("---")
print("-->{:>25}<--".format("rechts"))
print("---")
print("-->{:^25}<--".format("zentriert"))
```

Falls die Länge des Whitespace beim Zentrieren einer ungeraden Zahl von Zeichen entspricht, wird der rechte Whitespace entsprechend um ein Zeichen länger als der linke.

In [1]:
```
print("-->{:^29}<--".format("nicht so richtig zentriert"))
```

An Stelle von Whitespace können wir auch ein anderes Zeichen festlegen, mit dem dann der Whitespace gefüllt wird.

In [1]:
```
print("{:=^12}".format("> Aah! <"))
```

Das Padding Zeichen, hier `=` wird direkt hinter dem `:` und noch vor der Ausrichtung in der Formatspezifizierung angegeben.

Zahlen Formatierung

Auch Zahlen können mit Format-Strings besser dargestellt werden.

Grundsätzlich werden alle Nachkommastellen von Floats angezeigt, ...

In [1]:
```
print("Ohne Formatierung: {0}".format(1.23456789999999))
print("Achtung Rundungsfehler: {0}".format(1.2345678999999999999999))
```

... aber es kann zu Rundungsfehlern kommen.

Mit dem Zeichen `f` können wir explizit die Formatierung als Float erzwingen. Dann werden sechs Nachkommastellen ausgegeben und es wird entsprechend gerundet:

In [1]:
```
print("Als Float formatiert: {0:f}".format(1.23456789999999))
```

Wir können die Genauigkeit der Float-Ausgabe anpassen, indem wir die Anzahl der Nachkommastellen vorgeben:

In [1]:
```
print("Gerundet: {0:.4f}".format(1.23456789999999))
```

Der `.` und die darauf folgende Zahl (hier `4`) bestimmten die Anzahl der Dezimalen.

Wir können wie bei Strings eine Gesamtlänge für Zahlen festlegen und den führenden Whitespace mit Nullen füllen.

In [1]:
```
print("Mit festgelegter Gesamtlänge: {0:10.4f}".format(1.23456789999999))
print("Mit führenden Nullen: {0:010.4f}".format(1.23456789999999))
```

Außerdem können wir festlegen wir Vorzeichen dargestellt werden sollen. Per Default wird nur bei negativen Zahlen (notwendigerweise) das Vorzeichen angezeigt.

In [1]:
```
print("-->{}-->{}".format(100,-100))
print("-->{: d}-->{: d}".format(100,-100))
print("-->{:+d}-->{:+d}".format(100,-100))
```

Ein Leerzeichen nachdem Formatierungszeichen `:` sorgt dafür, dass ein Space für eventuelle negative Vorzeichen reserviert wird. Ein `+` an der gleichen Stelle erzwingt ein Vorzeichen

auch bei positiven Zahlen. Das Formatierungszeichen `d` erzwingt, dass der als Argument gegebene Wert als Dezimalzahl dargestellt wird.

Die Vorzeichendarstellung und Padding angaben können kombiniert werden. Das Vorzeichen wird dabei als ein Zeichen der Gesamtlänge des Ausdrucks berücksichtigt.

In [1]:
```
print("-->{: 10d}-->{: 10d}".format(100,-100))
print("-->{:+10d}-->{:+10d}".format(100,-100))
print("-->{:=10d}-->{:=10d}".format(100,-100))
print("-->{:=+10d}-->{:=+10d}".format(100,-100))
print("-->{:=+010d}-->{:=+010d}".format(100,-100))
```

Die Zahl vor dem Dezimalsystem-Symbol `d` bestimmt die wieviele Zeichen für die einzusetzenden Zahl reserviert wird. Mit dem `=` direkt nach dem `:` können wir beinflussen ob Vorzeichen links oder rechts vom Padding-Space ausgegeben werden. Wir können auch weiterhin führende Nullen erzwingen; dafür fügen wir vor der Längenangabe des Ausdrucks wieder eine `0` ein.

Formatierte Ausgabe anderer Datentypen

Falls wir Werte komplexer Datentypen ausgeben lassen wollen, müssen wir beachten, dass sich diese anders als Werte von Zahldatentypen und Strings nicht direkt mit den o.g. Werkzeugen formatieren lassen. Wenn wir die Werte **zunächst explizit in Strings umwandeln**, können wir sämtliche Formatierungsoptionen ebenfalls nutzen.

In [1]:
```
argument_list = [[1,2,3], "Listen", {1,2,3}, "Mengen", {1:"a", 2:"b"}, "Dic
format_string = "Formatierte Ausgabe von komplexen Datentypen:" + (3 * "\n

print(format_string.format(*argument_list))
```

Der Konversionsoperator `!` zusammen mit der Formatangabe `s` für String entspricht der Anwendung `str()` -Funktion auf den übergebenen Wert:

In [1]:
```
print("{:>20}".format(str([1,2,3])))
print("***")
print("{!s:>20}".format([1,2,3]))
```

Escapen von geschweiften Klammern in Format-Strings

Es kann vorkommen, dass wir in unserem Format-String geschweifte Klammern von der Interpretation als Platzhalter ausnehmen wollen. Durch das doppeln der Klammern `{{...}}` werden die inneren Klammern als normaler Text interpretiert:

In [1]:
```
print("Die geschweiften Klammern {{}} markieren {} und {}".format("Dictiona
```

Kaskadierende Format-Strings

Mit Hilfe von kaskadierenden Format-Strings können wir zum Beispiel Listen mit variabler Länge ausgeben lassen.

In [1]:
```
long_list = list(range(15))

pre_formatted_string = "{} " * (len(long_list)-1)  # Ein String mit Platzha
print(pre_formatted_string)
print("---")
formatted_string = "-->{}{{}}<--".format(pre_formatted_string)  # Wir wolle
print(formatted_string)
print("---")
print(formatted_string.format(*long_list))
print("---")
print("-->{}{{}}<--".format(pre_formatted_string).format(*long_list))
```

Wir wenden `format()` mehrmals hintereinander an um die Inhalte mit der gewünschten Formatierung auszugeben.

In [1]:
```
import random

def generate_three_word_sentences(nouns, verbs, n_sentences):
    for i in range(n_sentences):
        subject_object_list = random.sample(nouns, 2)
        verb = random.choice(verbs)
        pattern = "{} {{}} {}."  # Hier werden geschweifte Klammern escapet
        pattern_with_NPs = pattern.format(*subject_object_list)
        print(pattern_with_NPs)
        sentence = pattern_with_NPs.format(verb)
        print("{}\n---".format(sentence))


proper_nouns = ("Tom", "Kim", "Ashley", "Avery", "Elliott", "Jordan")
transitives = ("respects", "supports", "encourages", "appreciates", "admire

generate_three_word_sentences(proper_nouns, transitives, 3)
```

Hier ist eine Variante in der sich die Namenspaare und die benutzten Verben nicht wiederholen sollen.

In [1]:
```
import random

def pick_non_identical_samples(n_samples,n_samplesize, seq):
    picked_samples = []
    for i in range (n_samples):
        while True:
            new_sample = random.sample(seq, n_samplesize)
            if new_sample not in picked_samples:
                picked_samples.append(new_sample)
                break
    return picked_samples


def generate_non_identical_three_word_sentences(nouns, verbs, n_sentences):
    nouns_list = pick_non_identical_samples(n_sentences, 2, nouns)
    verb_list = pick_non_identical_samples(n_samples, 1, verbs)
    for i in range(n_sentences):
        subject_object_list = nouns_list[i]
        verb = verb_list[i]
        pattern = "{} {{}} {}."
        pattern_with_NPs = pattern.format(*subject_object_list)
        print(pattern_with_NPs)
        sentence = pattern_with_NPs.format(verb)
        print("{}\n---".format(sentence))


proper_nouns = ("Tom", "Kim", "Ashley", "Avery", "Elliott", "Jordan")
transitives = ("respects", "supports", "encourages", "appreciates", "admire

generate_non_identical_three_word_sentences(proper_nouns, transitives, 3)
```

Vorsicht! Abhängig von den übergebenen Argumenten kann es zu Probleme kommen.

F-Strings

Eine neuere Methode um Strings zu formatieren sind die **F-Strings** (ab Python 3.6, 2015). Sie sind noch etwas flexibler als die `format()` -Methode, die mit Python 3.0 2008 eingeführt wurden. F-Strings machen es noch einfacher Platzhalter mit Werten anderer Variablen zu füllen. Darüber hinaus werden F-Strings effizienter verarbeitet.

In [1]:
```
me = "Esther"
print(f"Huhu {name}!")
```

Der Python-Interpreter erkennt das Signal-Präfix `f` vor dem String, das den f-Strings ihren Namen gibt. Anschließend werden wie oben die Ausdrücke in geschweiften Klammern als Platzhalter mit Formatierungs Informationen verarbeitet. **Vorsicht! Die Platzhalter dürfen anders als bei Format-Strings nicht leer sein. Außerdem dürfen F-String-Platzhalter keine Backslashes `\` enthalten.**

Der Interpreter ist dabei auch in der Lage Variablennamen, Funktionen und andere Operationen zu erkennen, diese werden zunächst ausgewertet. Anschließend werden die resultierenden Werte, wenn möglich, zu Strings konvertiert und zuletzt werden alle Teil-Strings verbunden.

In [1]:
```
def ww():
    return "would walk"


text = ["miles", "And I"]

print(f"{text[1]} {ww()} {5*100} {text[0]} ...")
```

Wenn wir auf Dictionary-Elemente zugreifen wollen, müssen wir beachten die Keys als Strings zu markieren. Dies ermöglicht wiederum, dass die Key-Werte auch aus anderen Variablen gelesen werden können.

In [1]:
```
name_dict = {"title": "Dr", "first": "Angela Dorothea", "last": "Merkel"}

another_variable = "last"

print(f"{name_dict['last']}, {name_dict['title']} {name_dict['first']}")
print(f"{name_dict[another_variable]}, {name_dict['title']} {name_dict['fi
```

Vorsicht! Die Markierung der Key-Strings darf nicht die gleiche sein, die für den F-String benutzt wird. Deshalb werden im obigen Beispiel einfache `'...'` und doppelte Anführungszeichen `"..."` benutzt.

Auf die Platzhalter der F-Strings sind alle Formatierungen anwendbar, die wir für Format-Strings kennengelernt haben.

Zusammenfassung

- Format-Strings und F-Strings vereinfachen die Konvertierung und Ausgabe von Werten einfacher und komplexer Datentypen.
- Werte nahezu aller Datentypen können so einfach mit Hilfe von Platzhaltern in vorgefertigte Strings eingebettet und ausgegeben werden.
- Die Formatierungssyntax erlaubt es, Werte einheitlich und übersichtlich ausgeben zu lassen
- Die Unpacking Operatoren `*` für Sequenzen und `**` für Dictionaries ermöglichen es deren Elemente bzw. Schlüssel-Wert-Paare als einzelne Argumente bereitzustellen.
- Sowohl Format-Strings als auch F-Strings haben Eigenschaften, die unter bestimmten Umständen vorteilhaft gegenüber der jeweils anderen Formatierungsmethode sind.

Weitere Themen dieser Woche

- Der Datentyp Tuple