ist praktisch, wenn du in einer Interpreter-Sitzung auf das Ergebnis einer vorherigen Berechnung zugreifen möchtest:

```
>>> 20 + 3
23
>>> _
23
>>> print(_)
23
```

Der einzelne Unterstrich kommt auch sehr gelegen, wenn du Objekte im laufenden Betrieb konstruieren und mit ihnen arbeiten willst, ohne ihnen zuvor einen Namen zuzuweisen:

```
>>> list()
[]
>>> _.append(1)
>>> _.append(2)
>>> _.append(3)
>>> _
[1, 2, 3]
```

Kernpunkte

- **Führender einfacher Unterstrich: `_var`:** Bedeutet konventionsgemäß, dass der Name für den internen Gebrauch vorgesehen ist. Wird im Allgemeinen nicht vom Python-Interpreter durchgesetzt (außer bei Importen mit Sternchen) und ist nur als Hinweis für Programmierer gedacht.
- **Angehängter einfacher Unterstrich: `var_`:** Wird nach Konvention genutzt, um Namenskonflikte mit Python-Schlüsselwörtern zu vermeiden.
- **Führender doppelter Unterstrich: `__var`:** Löst im Kontext von Klassen eine Namensumformung aus. Wird vom Python-Interpreter umgesetzt.
- **Führender und angehängter doppelter Unterstrich: `__var__`:** Kennzeichnet besondere Methoden, die in Python definiert sind. Vermeide Namen dieser Art für deine eigenen Attribute.
- **Einzelner Unterstrich: `_`:** Wird manchmal als Name für temporäre oder unwichtige Variablen genutzt. Steht in Python-REPL-Sitzungen außerdem für das Ergebnis des letzten Ausdrucks.

2.5 Die furchtbare Wahrheit über Stringformatierung

In »The Zen of Python« heißt es, dass es »eine offensichtliche Möglichkeit« geben sollte, etwas zu tun. Umso erschütternder ist es dann festzustellen, dass es *vier* Vorgehensweisen für die Stringformatierung in Python gibt. In diesem Abschnitt führe ich diese vier Vorgehensweisen mit ihren jeweiligen Stärken und Schwächen

vor. Außerdem zeige ich dir eine einfache Faustregel, nach der ich die beste allgemeine Methode für die Stringformatierung auswähle.

Als einfaches Beispiel zur Veranschaulichung betrachten wir die folgenden Variablen (besser gesagt, Konstanten):

```
>>> errno = 50159747054
>>> name = 'Bob'
```

Aus diesen Variablen wollen wir nun einen Ausgabestring mit der folgenden Fehlermeldung zusammenstellen:

```
'Hey Bob, there is a 0xbadc0ffee error!'
```

Eine solche Meldung kann einem Entwickler wirklich den ganzen Tag versauen! Aber hier wollen wir uns um die Stringformatierung kümmern. Fangen wir an!

Stringformatierung auf die alte Weise

Für Strings gibt es in Python eine integrierte Operation, die über den Operator `%` zugänglich ist. Es handelt sich dabei um eine Kurzschreibweise zur Vereinfachung der positionsgestützten Formatierung. Wenn du schon einmal mit `printf`-Funktionen in C gearbeitet hast, sollte dir das Prinzip bekannt vorkommen. Betrachte das folgende Beispiel:

```
>>> 'Hello, %s' % name
'Hello, Bob'
```

Mit dem Formatspezifizierer `%s` teile ich Python mit, wo es den Wert von `name` als String einfügen soll. Dies ist die Stringformatierung im alten Stil.

Bei dieser Vorgehensweise kannst du auch noch andere Formatspezifizierer verwenden, um den Ausgabestring zu gestalten. Beispielsweise kannst du Zahlen in Hexadezimalschreibweise umwandeln oder Leerzeichen einfügen, um Tabellen und Berichte sauber zu formatieren.[11] So kann ich etwa mit dem Formatspezifizierer `%x` einen Integerwert in einen String umwandeln und als Hexadezimalzahl wiedergeben:

```
>>> '%x' % errno
'badc0ffee'
```

Wenn du in einem einzigen String mehrere Ersetzungen vornehmen möchtest, musst du die Syntax leicht abwandeln. Da der Operator `%` nur ein einziges Argument annimmt, musst du die rechte Seite wie folgt in ein Tupel stellen:

```
>>> 'Hey %s, there is a 0x%x error!' % (name, errno)
'Hey Bob, there is a 0xbadc0ffee error!'
```

11 Siehe Python-Dokumentation, »printf-style String Formatting«

Es ist auch möglich, in dem Formatstring namentlich auf die Ersetzungsvariablen zu verweisen, wenn du eine Zuordnung an den Operator `%` übergibst:

```
>>> 'Hey %(name)s, there is a 0x%(errno)x error!' % {
...     "name": name, "errno": errno }
'Hey Bob, there is a 0xbadc0ffee error!'
```

Dadurch lässt sich der Formatstring besser pflegen und leichter ändern. Du musst nicht darauf achten, dass du die Werte in der Reihenfolge übergibst, in der die Platzhalter in dem Formatstring stehen. Als Nachteil ergibt sich dabei allerdings mehr Tipparbeit.

Wahrscheinlich fragst du dich, warum diese Formatierung im `printf`-Stil als »Formatierung auf alte Weise« bezeichnet wird. Das liegt daran, dass sie technisch durch eine neue Vorgehensweise ersetzt wurde, die wir uns im nächsten Abschnitt ansehen. Die alte Variante steht zwar nicht mehr im Mittelpunkt, wurde aber noch nicht als veraltet und unerwünscht erklärt, sondern wird auch in den neuesten Versionen von Python noch unterstützt.

Formatierung auf die neue Weise

In Python 3 wurde eine neue Möglichkeit der Stringformatierung eingeführt und später auf Python 2.7 portiert. Dabei wird auf die Verwendung von `%`-Operatoren verzichtet und eine strenger reguläre Syntax verwendet. Die Formatierung erfolgt jetzt durch den Aufruf der Funktion `format()` für das Stringobjekt.[12] Mit `format()` kannst du ebenso wie mit der alten Vorgehensweise eine einfache positionsgestützte Formatierung durchführen:

```
>>> 'Hello, {}'.format(name)
'Hello, Bob'
```

Du kannst aber auch namentlich auf die Ersatzvariablen verweisen und sie in beliebiger Reihenfolge angeben. Das ist eine sehr vielseitige Vorgehensweise, da sie es ermöglicht, die Anzeigereihenfolge umzustellen, ohne die an die Formatierungsfunktion übergebenen Argumente zu ändern:

```
>>> 'Hey {name}, there is a 0x{errno:x} error!'.format(
...     name=name, errno=errno)
'Hey Bob, there is a 0xbadc0ffee error!'
```

In dem vorstehenden Beispiel siehst du auch, dass sich die Syntax zur Formatierung einer Integervariablen als Hexadezimalstring geändert hat. Jetzt übergeben wir eine *Formatspezifizierung*, indem wir das Suffix `:x` an den Variablennamen anhängen. Insgesamt hat sich die Formatstringsyntax als leistungsfähiger erwiesen, ohne die einfacheren Anwendungsfälle zu verkomplizieren. Es lohnt sich, diese *Minisprache zur Stringformatierung* in der Python-Dokumentation nachzulesen.[13]

12 Siehe Python-Dokumentation, »str.format()«

13 Siehe Python-Dokumentation, »Format String Syntax«

In Python 3 wird diese Stringformatierung der neuen Art gegenüber der Formatierung mit `%` bevorzugt. Seit Python 3.6 gibt es jedoch noch eine bessere Möglichkeit! Damit beschäftigen wir uns im nächsten Abschnitt.

Literalstringinterpolation (Python 3.6+)

Mit Python 3.6 kam eine weitere Möglichkeit zur Formatierung von Strings hinzu, nämlich *formatierte Stringliterale*. Dabei verwendest du eingebettete Python-Ausdrücke in Stringkonstanten. Betrachte dazu das folgende einfache Beispiel:

```
>>> f'Hello, {name}!'
'Hello, Bob!'
```

Diese neue Formatierungssyntax ist sehr vielseitig. Da sich beliebige Python-Ausdrücke einbetten lassen, kannst du damit sogar wie folgt Inline-Arithmetik durchführen:

```
>>> a = 5
>>> b = 10
>>> f'Five plus ten is {a + b} and not {2 * (a + b)}.'

'Five plus ten is 15 and not 30.'
```

Formatierte Stringliterale sind eine Eigenschaft des Python-Parsers, der solche `f`-Strings in eine Folge von Stringkonstanten und Ausdrücken umwandelt. Anschließend werden sie zum endgültigen String zusammengebaut.

Nehmen wir an, wir haben die folgende Funktion namens `greet()` mit einem `f`-String:

```
>>> def greet(name, question):
...     return f"Hello, {name}! How's it {question}?"
...

>>> greet('Bob', 'going')
"Hello, Bob! How's it going?"
```

Wenn wir diese Funktion disassemblieren und uns ansehen, was hinter den Kulissen vor sich geht, können wir erkennen, dass der `f`-String in der Funktion in etwas wie das Folgende umgewandelt wird:

```
>>> def greet(name, question):
...     return f"Hello, {name}! How's it {question}?"
...

>>> greet('Bob', 'going')
"Hello, Bob! How's it going?"
```

Die tatsächliche Implementierung ist ein wenig schneller, da sie den Opcode `BUILD_STRING` als Optimierung verwendet.[14] Von der Funktion her sind sie aber identisch:

```
>>> import dis
>>> dis.dis(greet)
  2           0 LOAD_CONST         1 ('Hello, ')
              2 LOAD_FAST          0 (name)
              4 FORMAT_VALUE       0
              6 LOAD_CONST         2 ("! How's it ")
              8 LOAD_FAST          1 (question)
             10 FORMAT_VALUE       0
             12 LOAD_CONST         3 ('?')
             14 BUILD_STRING       5
             16 RETURN_VALUE
```

Stringliterale können auch zusammen mit der Stringformatierungssyntax der Methode `str.format()` zum Einsatz kommen. Damit kannst du die Formatierungsaufgaben, die wir in den letzten beiden Abschnitten besprochen haben, ebenfalls lösen:

```
>>> f"Hey {name}, there's a {errno:#x} error!"
"Hey Bob, there's a 0xbadc0ffee error!"
```

Die neuen formatierten Stringliterale in Python ähneln den JavaScript-Template-Literalen, die in ES2015 hinzugefügt wurden. Ich halte sie für eine gute Ergänzung der Sprache und habe sie bereits in meiner täglichen Arbeit mit Python 3 verwendet. Um mehr über formatierte Stringliterale zu erfahren, schlage in der offiziellen Python-Dokumentation nach.[15]

Template-Strings

Es gibt noch eine weitere Technik zur Stringformatierung in Python, nämlich die Verwendung von Template-Strings. Das ist ein einfacherer und weniger vielseitiger Mechanismus, aber in einigen Fällen kann er genau das sein, was du gerade brauchst. Sieh dir das folgende einfache Beispiel einer Begrüßung an:

```
>>> from string import Template
>>> t = Template('Hey, $name!')
>>> t.substitute(name=name)
'Hey, Bob!'
```

Wie du siehst, müssen wir hierzu die Klasse `Template` aus dem integrierten Python-Modul `string` importieren. Template-Strings gehören nicht zu den Kernmerkmalen der Sprache, sondern werden durch ein Modul in der Standardbibliothek bereitgestellt.

14 Siehe *https://bugs.python.org/issue27078*

15 Siehe Python-Dokumentation, »Formatted String Literals«

Ein weiterer Unterschied besteht darin, dass in Template-Strings keine Formatspezifizierer zulässig sind. Damit unser Beispiel mit der Fehlermeldung funktioniert, müssen wir die Nummer des Fehlers selbst in einen Hexstring umwandeln:

```
>>> templ_string = 'Hey $name, there is a $error error!'
>>> Template(templ_string).substitute(
...     name=name, error=hex(errno))
'Hey Bob, there is a 0xbadc0ffee error!'
```

Das funktioniert zwar großartig, aber vielleicht frage dich trotzdem, wann du in deinen Python-Programmen Template-Strings verwenden sollst. Meiner Meinung nach sind sie am besten für den Umgang mit Formatstrings geeignet, die von den Benutzern des Programms bereitgestellt werden. Aufgrund ihrer geringeren Komplexität sind sie sicherer.

Die kompliziertere Formatierungs-Minisprache der anderen Techniken kann in deinen Programmen Sicherheitslücken öffnen. Beispielsweise können Formatstrings in deinen Programmen auf willkürliche Variablen zugreifen. Wenn ein böswilliger Benutzer einen Formatstring bereitstellt, kann er damit geheime Schlüssel und andere sensible Informationen gewinnen. Das folgende einfache Beispiel zeigt, wie ein solcher Angriff im Prinzip ablaufen kann:

```
>>> SECRET = 'this-is-a-secret'
>>> class Error:
...     def __init__(self):
...         pass
>>> err = Error()
>>> user_input = '{error.__init__.__globals__[SECRET]}'

# Oweia ...
>>> user_input.format(error=err)
'this-is-a-secret'
```

Hier konnte der Angreifer den geheimen String abrufen, indem er von dem Formatstring aus auf das Dictionary `__globals__` zugriff. Erschreckend, nicht wahr? Bei Template-Strings besteht diese Angriffsmöglichkeit nicht, weshalb sie sicherer sind, wenn du aus Benutzereingaben generierte Formatstrings verarbeiten musst:

```
>>> user_input = '${error.__init__.__globals__[SECRET]}'
>>> Template(user_input).substitute(error=err)
ValueError:
"Invalid placeholder in string: line 1, col 1"
```

Welche Methode zur Stringformatierung solltest du verwenden?

Bei so vielen Möglichkeiten zur Stringformatierung in Python hast du die Qual der Wahl. Vielleicht sollte ich dir jetzt ein Flussdiagramm zur Entscheidungsfindung zeigen. Aber das werde ich nicht tun. Stattdessen stelle ich dir die einfache Faustregel vor, die ich beim Schreiben von Python-Code anwende. Wende einfach

- **Du möchtest numerische Daten speichern (Integer oder Fließkommazahlen), wobei eine effiziente Speichernutzung und die Leistung sehr wichtig sind?** Probiere aus, ob dir `array.array` alles bietet, was du brauchst. In einem solchen Fall kannst du dich auch außerhalb der Standardbibliothek umsehen und Pakete wie `NumPy` oder `Pandas` verwenden.
- **Du hast Textdaten in Form von Unicode-Zeichen?** Verwende den integrierten Python-Datentyp `str`. Wenn du einen veränderbaren String brauchst, nimm eine Liste aus Zeichen.
- **Du willst einen zusammenhängenden Block von Bytes speichern?** Verwende den unveränderbaren Typ `bytes`. Wenn du eine veränderbare Datenstruktur benötigst, nimm `bytearray`.

In den meisten Fällen nehme ich zu Anfang eine einfache Liste und wechsle später zu einem spezielleren Typ, falls die Leistung oder der Speicherplatz eine Rolle spielen. Meistens bietet eine Allzweck-Arraydatenstruktur wie `list` die höchste Entwicklungsgeschwindigkeit und auch den größten Komfort zum Programmieren. Meiner Erfahrung nach ist das zu Anfang viel wichtiger, als gleich zu Beginn zu versuchen, auch noch das letzte Quäntchen Leistung herauszuholen.

5.3 Datensätze, Strukturen und DTOs

Datensätze enthalten eine feste Anzahl von Feldern, die jeweils einen eigenen Namen haben und auch einen eigenen Typ aufweisen können. In diesem Abschnitt sehen wir uns an, wie du in Python Datensätze, Strukturen und einfache, klassische Datenobjekte mit den integrierten Typen und den Klassen aus der Standardbibliothek implementierst. Übrigens verwende ich den Begriff »Datensatz« (*record*) hier in einer sehr weit gefassten Bedeutung. Beispielsweise bespreche ich hier auch Typen wie Tupel, die nicht unbedingt Datensätze im strengen Sinne sind, da sie keine benannten Felder enthalten. Python stellt verschiedene Datentypen zur Implementierung von Datensätzen, Strukturen und Datenübertragungsobjekten (Data Transfer Objects, DTOs) bereit. In diesem Abschnitt sehen wir uns die einzelnen Implementierungen und ihre jeweiligen Eigenschaften an. Am Ende erhältst du eine Zusammenfassung und einen Leitfaden für die Auswahl.

Fangen wir an!

Einfache Datenobjekte: dict

Python-Dictionarys können eine beliebige Anzahl von Objekten enthalten, die jeweils durch einen eindeutigen Schlüssel identifiziert werden.[15] Sie werden auch als *Maps* und *assoziative Arrays* bezeichnet und ermöglichen das effiziente Nachschlagen, Einfügen und Löschen beliebiger Objekte, die mit einem Schlüssel verknüpft sind. Es ist möglich, Dictionarys als Datentyp oder Datenobjekt für Datensätze zu verwenden. Sie lassen sich leicht erstellen, da eine gewisse syntaktische

15 Siehe Abschnitt »Dictionarys«

```
    mileage: float
    automatic: bool

>>> car1 = Car('red', 3812.4, True)

# Instanzen haben eine übersichtliche Darstellung:
>>> car1
Car(color='red', mileage=3812.4, automatic=True)

# Zugriff auf Felder:
>>> car1.mileage
3812.4

# Felder sind unveränderbar:
>>> car1.mileage = 12
AttributeError: "can't set attribute"
>>> car1.windshield = 'broken'
AttributeError:
"'Car' object has no attribute 'windshield'"

# Typanmerkungen werden ohne ein zusätzliches Typüberprüfungswerkzeug
# wie mypy nicht durchgesetzt:
>>> Car('red', 'NOT_A_FLOAT', 99)
Car(color='red', mileage='NOT_A_FLOAT', automatic=99)
```

Serialisierte C-Strukturen: struct.Struct

Die Klasse `struct.Struct` wandelt Python-Werte in serialisierte C-Strukturen aus `bytes`-Objekten um.[23] Damit kannst du beispielsweise Binärdaten handhaben, die in Dateien gespeichert sind oder über Netzwerkverbindungen hereinkommen. Solche Strukturen werden mit einer Minisprache ähnlich wie für Formatstrings definiert, mit der du die Anordnung verschiedener C-Datentypen wie `char`, `int` und `long` sowie ihrer vorzeichenlosen Varianten (`unsigned`) festlegen kannst. Serialisierte Strukturen werden nur selten für Datenobjekte verwendet, die ausschließlich im Python-Code gehandhabt werden sollen. Sie dienen hauptsächlich als Datenaustauschformat und nicht als eine Möglichkeit, um Daten, die nur im Python-Code verwendet werden, im Arbeitsspeicher festzuhalten.

Wenn du primitive Daten in Strukturen verpackst, kannst du damit manchmal weniger Arbeitsspeicher verbrauchen als bei der Verwendung anderer Datentypen. Meistens ist eine solche Optimierung jedoch ziemlich anspruchsvoll und wahrscheinlich unnötig.

```
>>> from struct import Struct
>>> MyStruct = Struct('i?f')
>>> data = MyStruct.pack(23, False, 42.0)
```

23 Siehe Python-Dokumentation, »struct.Struct«