Scope

Geltungsbereich von Variablen

Scope Definition

In der Programmierung definiert der Scope (deutsch: Geltungsbereich) den Bereich, in dem auf einen Bezeichner eindeutig zugegriffen werden kann, wie z. B. Variablen, Funktionen, Objekte usw.

Ein Bezeichner ist nur in seinem Geltungsbereich sichtbar und zugänglich.

Programmiersprachen machen sich den Geltungsbereich zunutze, um Namenskollisionen und unvorhersehbare Verhaltensweisen zu vermeiden.

Hintergrund

Um mit globalen Namen zu arbeiten, musste man den gesamten Code gleichzeitig im Auge behalten, um zu wissen, welchen Wert ein bestimmter Name zu einem bestimmten Zeitpunkt hat.

Scopes versuchen, diesen immensen Nachteil zu umgehen.

Arten von Scopes

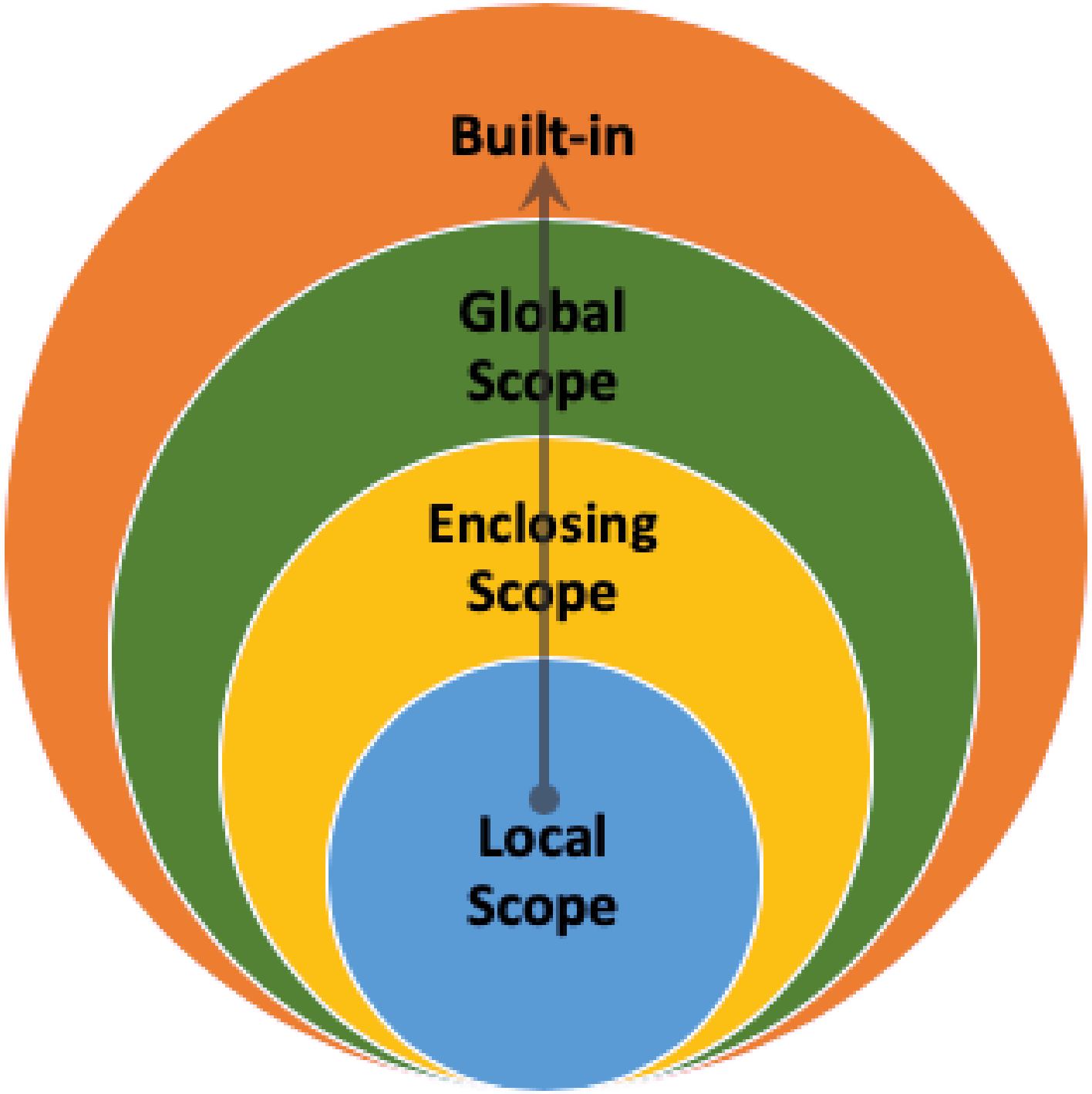
Grundsäztlich unterscheiden wir zwei wichtige Arten von Scopes, den lokalen Scope und den globalen Scope. Später werden wir noch weitere Scopes kennenlernen.

Globaler Scope:

Die Namen, die in diesem Bereich definiert werden, sind überall im Code, zumindest lesend, verfügbar.

Lokaler Scope:

Die Namen, die in diesem Bereich definiert werden, sind nur für den Code innerhalb dieses Bereichs verfügbar oder sichtbar.



Die verschiedenen Scopes in Python

Der weiteste Scope ist der built-in Scope, er ist von überall erreichbar.

Der globale Scope ist überall im Programm erreichbar.

Der Enclosing Scope ist ähnlich dem globalen Scope, aber für verschachtelte Funktionen.

Der locale Scope (auch Funktionsscope) ist der Scope innerhalb Funktionen.

Funktionsweise: Die LEGB-Regel

Der Geltungsbereich eines Bezeichners (Scope) bestimmt seine Sichtbarkeit im Code. In Python ist der Geltungsbereich entweder als lokaler, umschließender, globaler oder built-in Geltungsbereich implementiert.

Wenn wir nun eine Variable oder Funktion verwenden, durchsucht Python diese Bereiche der Reihe nach, um sie aufzulösen.

Wenn der Name nicht gefunden wird, erhalten wir eine Fehlermeldung. Dies ist der allgemeine Mechanismus, den Python zur Namensauflösung verwendet und der als LEGB-Regel bekannt ist.

Lokaler Scope

Der lokale Scope ist der Körper einer Python-Funktion oder eines Lambda-Ausdrucks. Dieser Python-Bereich enthält die Namen, die innerhalb der Funktion definiert werden. Namen, die innerhalb dieses Scopes definiert werden, sind außerhalb nicht verfügbar.

```
def fn():
    x = 2
print(x) => Bezeichner x ist unbekannt.
```

Die built-in Funktion locals()

Die eingebaute Funktion locals() zeigt alle Variablen, die im lokalen Scope verfügbar sind, an. Die Variable p ist im globalen Raum definiert und keine lokale Variable.

```
p = 3.3

def fn(a):
    x = 2
    print(locals())

fn(4)
```

```
{'a': 4, 'x': 2}
```

Globaler Scope

Der globale Scope ist der oberste Bereich in einem Python-Programm, -Skript oder -Modul. Dieser Python-Bereich enthält alle Namen, die auf der obersten Ebene eines Programms oder eines Moduls definiert werden.

Bezeichner in diesem Python-Bereich sind von überall im Code sichtbar.

```
RADIUS = 2

def fn():
    y = 3 + RADIUS
    print(y)
```

Die built-in Funktion globals()

Die eingebaute Funktion globals() zeigt alle Variablen, die im globalen Scope verfügbar sind, an.

```
p = 3.3

def fn(a):
   x = 3

fn(4)

print(globals())
```

```
{'p': 22.22, 'fn': <function fn at 0x0000026A408C9AF0>}
```

Lokaler Scope: Variablenzuweisungen

Wenn der Python Compiler eine Variablenzuweisung in einer Funktion/Methode (lokaler Scope) sieht, wird dieser den Name automatisch als lokal markieren und daher keine ähnlich benannten Variablen außerhalb berücksichtigen.

Wenn er also sieht, dass vor der Zuweisung der lokalen Variable diese innerhalb der Funktion für etwas anderes verwendet wird, wird er einen Fehler ausgeben, dass versucht wird, eine Variable zu verwenden, die noch nicht referenziert wurde. Im Beispiel unten wird amount von Python lokal gemacht und dann wird versucht, diese lokale Variable mit 1 zu addieren. Das geht nicht und es kommt zu einem UnboundLocalError.

```
items = [2]
amount = 1

def fn():
    items.append(3)  # 3 wird in Liste items angehängt. Es findet KEINE Variablenzuweisung statt.
    amount_neu = amount + 1 # amount_neu wird das Ergebnis von globale amount + 1 zugewiesen
    amount  = amount + 1   # UnboundLocalError, da versucht wird, an lokale amount 1 zu addieren
    amount += 1   # UnboundLocalError, da versucht wird, an lokale Variable amount 1 zu addieren
```

Lokaler Scope: das global Keyword

Mit dem global Keyword lassen sich innerhalb von Funktionen (lokaler Scope) globale Variablen definieren. Falls ein Bezeichner dieser Art global schon existiert, wird dieser genutzt. Somit kann innerhalb Funktionen eine Zuweisung an eine globale Variable genutzt werden.

```
counter = 0  # global definiert
def update_counter():
    global visit
    global counter  # innerhalb von Funktion counter als global definieren
    counter += 1  # globale Variable counter um 1 erhöhen
    visit = 3  # globale Variable

update_counter()
update_counter()
counter
```

2

```
# Zugriff auf (jetzt globale) Variable visit
visit
```

3

das global Keyword

Grundsätzlich gilt der Einsatz des global Keywords als bad practice. Es fördert Seiteneffekte und verschlechtert die Les- und Testbarkeit einer Funktion. Funktionen sollten möglichst isoliert sein und keine globalen Variablen verändern.

Im Beispiel ein Seiteneffekt, der durch den Aufruf von set_secret_key ausgeführt wird:

```
secret_key = 0

def set_secret_key(value: int) -> None:
    global secret_key
    secret_key = (value ** 2) // 3

set_secret_key(2243324)
print(secret_key)
```

Statt global expliziter Return

Um im Kontrollfluss von großen Programmen den Überblick zu behalten, ist es besser, wenn Funktionen keine Auswirkungen auf die Umwelt haben.

```
def get_secret_key(value: int) -> int:
    return value ** 2 // 3


secret_key = get_secret_key(2243324)
print(secret_key)
```

Built-In Scope

Wenn der Bezeichner weder im lokalen, globalen oder enclosing Scope gefunden wird, beginnt Python, im Built-In Scope nach ihm zu suchen.

Dieser deckt alle Bezeichner der eingebauten Funktionen ab. Sie können vor ihrer Verwendung überall im Programm aufgerufen werden, ohne dass sie definiert oder importiert werden müssen.

Eingebaute Funktion print:

print(„call me without importing me“)

Built-In Funktionen überschreiben

Ein häufiger Fehler und eine bad practice ist es, eingebaute Funktionen zu überschreiben. In Python ist das möglich aber sollte vermieden werden.

Im Bespiel weisen wir dem Bezeichner id den Wert 3 zu. Später im Code wollen wir die built-in Funktion id ausführen, um die Identität eines Integers zu prüfen.

```
id = 3

# Ausführen der Funktion id, um die Identität eines Integers zu prüfen. Nur ist id jetzt ein Integer.
id(42)
```

=> Fehler. TypeError: 'int' object is not callable

Enclosing Scope

Das Schlüsselwort nonlocal wird ausschließlich in verschachtelten Funktionen verwendet, um auf eine Variable in der übergeordneten Funktion zu verweisen. Es funktioniert also ähnlich wie global, nur eben auf den übergeordneten Scope einer verschachtelten Funktion bezogen. Der Einsatz von nonlocal ist eher selten und sei daher nur am Rande erwähnt.

Eine Funktion outer definiert eine Variable x und eine Funktion inner. In der Funktion inner wird nun x auf nonlocal gesetzt und kann verändert werden. Ohne das nonlocal Keyword würde es zu einem Fehler kommen.

```
def outer():
    x = 3
    y = 42
    def inner():
        nonlocal x
        x += 1
        y = 7  # hier wird eine neue lokale Variable y eingeführt
    inner()
    print(x)
    print(y)

outer()
```

```
4
42
```