Python

Akihiro Minamino

September 2, 2020

1　リテラル

リテラルとは、Python のプログラムに直接記述された値です。以下の種類がある。

1.1　組み込み定数

- True: 真を表す bool 型の値。整数に変換すると 1 になる。
- Flase: 偽を表す bool 型の値。整数に変換すると 0 になる。
- None: 存在しないことを表す NoneType 型の値。
- その他の組み込み定数（NotImplemented など）

1.2　文字列リテラル

- シングルクォーテーションまたはダブルクォーテーションで囲む。改行を直接含めたいといは、三重引用符（三重シングルクォーテーションまたは三重ダブルクォーテーション）を使う。

1.3　数値リテラル

- 整数や浮動小数点数などの数字。

1.4　その他

- リスト
- タプル
- 辞書
- 集合

2　変数

2.1　変数、名前、オブジェクト

Python 変数の重要なポイントは、変数はただの名前だということである。データをいれているオブジェクトに名前を付けるだけである。名前は値自体ではなく値の参照である。名前は、オブジェクトに貼るポストイットのようなものである。

2.2　数値

Python の数字の並びは、リテラル[1]の整数と見なされる。

Python では、=記号の右辺の式がまず計算され、次に左辺の変数に代入が行われる。

2.3　基数

整数は、プレフィックスで基数を指定しない限り、10 進（基数 10）と見なされる。基数は、「桁上り」しなければならなくなるまで、何個の数字を使えるかを示す。
Python では、10 進以外に 3 種類の基数を使ってリテラル整数を表す。

- 0b は 2 進（基数 2）
- 0o は 8 進（基数 8）
- 0x は 16 進（基数 16）

インタープリターは、10 進整数として、整数を表示する。

```
1 >>> 10
2 10
3 >>> 0b10
4 2
5 >>> 0x10
6 16
```

2.4　型の変換

Python の整数以外のデータ型を整数に変換するには、int() 関数を使う。この関数は整数部だけを残し、小数部を切り捨てる。

int() は、数字でできた文字列を整数に変換する。しかし、小数点や指数部を含む文字列は処理しない。

[1] リテラルとは、プログラムのソースコードにおいて使用される、数値や文字列を直接に記述した定数のことである。変数の対義語であり、変更されないことを前提とした値である。

```
>>> int('98.6')
ValueError: ...
>>> int('1.0e4')
ValueError: ...
```

他のデータ型の値を float に変換するには、`float()` 関数を使う。

```
>>> float(True)
1.0
>>> float(False)
0.0
>>> float(98)
98.0
>>> float('99')
99.0
>>> float('98.6')
98.6
>>> float('-1.5')
-1.5
>>> float('1.0e4')
10000.0
```

2.5 文字列

文字列は、文字のシーケンスである。

他の言語と異なり、Python の文字列はイミュータブルである。つまり、文字列をその場で書き換えることができない。

2.5.1 クォートを使った作成

Python 文字列は、シングルクォートかダブルクォートで文字を囲んで作る。どちらのクォートを使っても、Python はまったく同じように扱う。2 種類のクォート文字を使えるようにしている理由は、クォート文字を含む文字列を作りやすくするためである。ダブルクォートで文字列を作るときは、文字列内にシングルクォートを入れることができ、シングルクォートで文字列を作るときは、文字列内にダブルクォートを入れることができる。

```
>>> "Nay,' said the naysayer."
"Nay,' said the naysayer."
```

```
>>> 'The rare double quote in captivity: ".'
'The rare double quote in captivity: ".'
>>> 'A "two by four" is actually 1 1/2" X 3 1/2".'
'A "two by four" is actually 1 1/2" X 3 1/2".'
>>> "'There's the man that shot my paw!' cried the limping
    hound."
"'There's the man that shot my paw!' cried the limping hound."
```

3個のシングルクォート（'''）や3個のダブルクォート（"""）を使うこともできる。

```
>>> '''Boom!'''
'Boom'
>>> """Eek!"""
'Eek!'
```

トリプルクォートは、次のような複数行文字列を作るために使われる。

```
>>> poem = '''There was a Young Lady of Norway,
... Who casually sat in a doorway;
... When the door squeezed her flat,
... She exclaimed, "What of that?"
... This courageous Young Lady of Norway.'''
>>> print(poem)
There was a Young Lady of Norway,
Who casually sat in a doorway;
When the door squeezed her flat,
She exclaimed, "What of that?"
This courageous Young Lady of Norway.
```

トリプルクォートのなかに複数行の文字列を入れると、その文字列には改行文字も残される。先頭や末尾にスペースがある場合、それらも残る。

```
>>> poem2 = '''I do not like thee, Doctor Fell.
...      The reason why, I cannot tell.
...      But this I know, and know full well:
...      I do not like thee, Doctor Fell.
... '''
>>> print(poem2)
I do not like thee, Doctor Fell.
    The reason why, I cannot tell.
    But this I know, and know full well:
    I do not like thee, Doctor Fell.

```

```
>>> poem2
'I do not like thee, Doctor Fell.\n    The reason why, I
   cannot tell.\n    But this I know, and know full well:\n
      I do not like thee, Doctor Fell.\nprint(poem2)\n'
```

print() は文字列からクォートを取り除き、表示する項目の間にスペースを追加し、末尾に改行を追加し、文字列の内容を表示する。
文字列には空文字列がある。文字がひとつも含まれていない文字列だが、完全に有効な文字列として扱われる。空文字は、クォートを使って以下のように作る。

```
>>> ''
''
>>> ""
''
>>> ''''''
''
>>> """""""
''
>>>
```

空文字は、他の文字列から新しく文字列を組み立てたいときに、まず白紙のノートが必要になる、

```
>>> bottles = 99
>>> base = ''
>>> base += 'current inventory: ' #在庫
>>> base += str(bottles)
>>> base
'current inventory: 99'
```

2.5.2 str() を使った型変換

str() 関数を使うと、他のデータ型を文字列に変換できる。

```
>>> str(98.6)
'98.6'
>>> str(1.0e4)
'10000.0'
>>> str(True)
'True'
```

2.5.3 \ によるエスケープ

特定の文字の前にバックスラッシュ（\）を入れると、特別な意味になる。もっともよく使われるエスケープシーケンスは、改行の意味になる\nだ。

```
>>> palindrome = 'A man,\nA plan,\nA canal:\nPanama.'
>>> print(palindrome)
A man,
A plan,
A canal:
Panama.
```

テキストの位置を揃えるために\tというエスケープシーケンスもよく使う。

```
>>> print('\tabc')
    abc
>>> print('a\tbc')
a   bc
>>> print('ab\tc')
ab  c
>>> print('abc\t')
abc
```

最後の文字列の末尾にタブ文字が含まれるが、目には見えない。

文字列を囲むために使っているシングルクォート、ダブルクォートを文字列内でもリテラルとして使いたい場合は、\'と\"が必要になる。

```
>>> testimony = "\"I did nothing!\" he said. \"Not that either
   ! Or the other thing.\""
>>> print(testimony)
"I did nothing!" he said. "Not that either! Or the other thing
   ."
>>> fact = "The world's largest rubber duck was 54'2\" by
   65'7\" by 105'"
>>> print(fact)
The world's largest rubber duck was 54'2" by 65'7" by 105'
```

リテラルのバックスラッシュが必要な場合には、バックスラッシュをふたつ重ねる。

```
>>> speech = 'Today we honor our friend, the backslash: \\.'
>>> print(speech)
Today we honor our friend, the backslash: \.
```

2.5.4 +による連結

Python では、+演算子を使えば、リテラル文字列、文字列変数を連結できる。

```
>>> 'Release the kraken! ' + 'At once!'
'Release the kraken! At once!'
```

リテラル文字列の場合は、順に並べるだけでも連絡できる（文字列変数はできない)。

```
>>> "My word! " "A gentleman caller!"
'My word! A gentleman caller!'
```

文字列の連結では、Python は自動的にスペースを追加しない。それに対し、print() 関数は、各引数の間にはスペースを挿入し、末尾に改行を追加する。

```
>>> a = 'Duck.'
>>> b = a
>>> c = 'Grey Duck!'
>>> a + b + c
'Duck.Duck.Grey Duck!'
>>> print(a, b, c)
Duck. Duck. Grey Duck!
```

2.5.5 *による繰り返し

*演算子を使うと、文字列を繰り返すことができる。

```
>>> a = 'Duck.'
>>> start = 'Na ' * 4 + '\n'
>>> middle = 'Hey ' * 3 + '\n'
>>> end = 'Goodbye.'
>>> print(start + start + middle + end)
Na Na Na Na
Na Na Na Na
Hey Hey Hey
Goodbye.
```

2.5.6 [ ] による文字の抽出

文字列のなかのひとつの文字を取り出したいときは、文字列名の後ろに [ ] で囲んだ文字のオフセットを各。先頭の文字（もっとも左）のオフセットは 0、その右が 1 と数える。末尾（もっとも右）の文字のオフセットは-1 えも指定できる。右端の左は-2、されにその左は-3 のように続く。

```
>>> letters = 'abcdefghijklmnopqrstuvwxyz'
>>> letters[0]
'a'
>>> letters[1]
'b'
>>> letters[-1]
'z'
>>> letters[-2]
'y'
>>> letters[25]
'z'
>>> letters[5]
```

文字列の長さ以上のオフセットを指定すると、例外が起きる。

```
>>> letters[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module
```

このインデックス参照は、他のシーケンス型（リストやタプル）でも機能する。

文字列はイミュータブルなので、文字列に直接文字を挿入したり、指定したインデックスの位置の文字を書き換えたりすることはできない。

2.5.7 [start:end:step] によるスライス

スライスを使えば、文字列から部分文字列（文字列の一部）を取り出すことができる。スライスは、[ ] と先頭オフセット（start)、末尾オフセット（end)、ステップ（step）で定義する。

- `[ : ]` は、先頭から末尾までのシーケンス全体を抽出する。
- `[ start: ]` は、`start` オフセットから末尾までのシーケンスを抽出する。
- `[ :end ]` は、先頭から `end-1` オフセットまでのシーケンスを抽出する。
- `[ start:end ]` は、`start` オフセットから `end-1` オフセットまでのシーケンスを抽出する。
- `[ start:end:step ]` は、`step` 文字毎に `start` オフセットから `end-1` オフセットまでのシーケンスを抽出する。

```
>>> letters = 'abcdefghijklmnopqrstuvwxyz'
>>> letters[ : ]
```

```
'abcdefghijklmnopqrstuvwxyz'
>>> letters[20:]
'uvwxyz'
>>> letters[10:]
'klmnopqrstuvwxyz'
>>> letters[12:15]
'mno'
>>> letters[-3:]
'xyz'
>>> letters[18:-3]
'stuvw'
>>> letters[-6:-3]
'uvw'
>>> letters[::7]
'ahov'
>>> letters[4:20:3]
'ehknqt'
>>> letters[19::4]
'tx'
>>> letters[:21:5]
'afkpu'
```

Python では、末尾の指定は実際のオフセットよりもひとつ先でなければならない。

ステップサイズとして負数を指定すると、この便利なスライスは逆にステップしていく。

```
>>> letters = 'abcdefghijklmnopqrstuvwxyz'
>>> letters[-1::-1]
'zyxwvutsrqponmlkjihgfedcba'
>>> letters[::-1]
'zyxwvutsrqponmlkjihgfedcba'
```

2.5.8 len() による長さの取得

Python の組み込み関数 `len()` は、文字列内の文字数を数える。

```
 >>> letters = 'abcdefghijklmnopqrstuvwxyz'
>>> len(letters)
26
>>> empty = ''
>>> len(empty)
0
```

2.5.9 split() による分割

文字列専用関数を使うときには、文字列の名前をタイプしてからドット、さらに関数名をタイプし、関数が必要とする引数を指定する。つまり、string.function(arguments) という形式である。
文字列関数 split() は、セパレータに基づいて文字列を分割し、部分文字列のリストを作る。

```
1 >>> todos = 'get gloves, get mask, get cat vitamins, call
      ambulance'
2 >>> todos.split(',')
3 ['get gloves', ' get mask', ' get cat vitamins', ' call
      ambulance']
```

セパレータを指定していない split() は、セパレーターとして空白文字 (改行、スペース、タブ) のシーケンスを使う。

```
1 >>> todos.split()
2 ['get', 'gloves,', 'get', 'mask,', 'get', 'cat', 'vitamins,',
      'call', 'ambulance']
```

Python は括弧の有無で関数呼び出しかどうかを判断しているので、引数なしで split を呼び出すときでも括弧は必要。

2.6 join() による結合