第１章

Pythonをはじめよう！

さあ、Pythonの学習を始めましょう。本章では、Pythonの特徴などを理解するとともに、Pythonの基礎を学習します。

- Pythonとは
- Pythonプログラムの実行
- インタラクティブシェル（基本対話モード）
- 演算子とオペランド
- 単項演算子と２項演算子
- 基本的な算術演算子
- 演算子の優先度
- 型とtype関数
- 数値型
- 整数型（int型）／浮動小数点数型（float型）／複素数型（complex型）
- 数値リテラル
- 整数リテラル（２進／８進／10進／16進）と浮動小数点数リテラル
- 文字列
- 文字列リテラルと原文字列リテラル
- エスケープシーケンス
- 変数
- 代入文
- \による行の継続
- Pythonの哲学（The Zen of Python）

1-1 Pythonとは

さあ、Pythonの学習を始めましょう。まずは、Pythonの特徴や、歴史的変遷などを学習していきます。

Pythonについて

コンピュータが処理を行っているときは、何らかの**プログラム**が動いています。そのプログラムとは、**コンピュータを動作させるための命令**の集まりです。

プログラム記述のために体系化されたのが、**プログラミング言語**（*programming language*）と呼ばれる人工言語です。

これから学習する**Python**は、オランダ出身のグイド・ヴァン・ロッサム（*Guido van Rossum*）氏が開発したプログラミング言語です。名前の由来は、イギリスのBBCが製作したコメディ番組「**空飛ぶモンティ・パイソン**（*Monty Python's Flying Circus*）」に由来します。

▶ 英語の名詞としてのpythonは、『ニシキヘビ』という意味です。そのため、Pythonのロゴマークではヘビの絵が使われています（**Fig.1-1**）。

現在、Pythonの開発・維持は、**Pythonソフトウェア財団**（*Python Software Foundation*）で行われています。私たちが、Pythonの情報を得るための定番ともいえるのが、以下の三つのサイトです。

Ⓐ Pythonの基本的な情報：Pythonソフトウェア財団

`https://www.python.org/`

Pythonのダウンロードなどは、このページから行います。

Ⓑ 日本語の情報：日本Pythonユーザ会（PyJUG）

`https://www.python.jp/`

日本Pythonユーザ会のホームページです。日本語での情報発信が行われています。

Ⓒ 日本語のドキュメント（Python Ver.3）

`https://docs.python.org/ja/3/`

チュートリアル（入門）、言語リファレンス、ライブラリーリファレンス、FAQ（よくある質問と、その回答）など、Ⓐの多くのドキュメント類が日本語に翻訳されて公開されています。

▶ もともとⒷのサイトの一部として公開されていましたが、現在はⒶのサイトの一部となっています。

Fig.1-1 Pythonのロゴマーク

Python の特徴

プログラミング言語の一つである Python の特徴を眺めていきましょう。なお、専門用語などを使って解説していますので、現時点では、理解できない箇所があっても構いません。

▪ フリーのオープンソースソフトウェアである

Python は、**無料**で使えます。しかも、その**ソース**までもが公開されています（Python 自体が、どのように作られているのかが公開されています）。そのため、Python を使ってプログラムを作るだけでなく、Python 自身の中身まで調べたり学習したりできます。

▪ マルチプラットフォームである

MS-Windows、macOS、Linux など、多くの環境で動作します。

▪ 各種ドキュメントが豊富

チュートリアルを含むドキュメントがインターネット上で公開されています。

▪ 幅広い分野に適用できる汎用言語である

すべてのプログラミング言語が、あるゆる分野に適用できるわけではありません。数値計算が得意な言語、事務処理やデータベースが得意な言語、といった具合で、言語ごとに得意分野があります。

その意味で、Python は、オールラウンドプレーヤー的な存在です。

機械学習、**ディープラーニング**などの **AI**（**人工知能**）**分野**、**データ解析**、**科学技術計算**、**Web アプリケーション**、**GUI**（***graphical user interface***）など、極めて多くの分野を得意としています。

しかも、他の言語で開発したプログラムとの組合せが容易な“**グルー言語**（接着剤のような言語という意味です）”としての性格をもちますので、Python が不得意な分野では、それを得意とする他の言語で開発したプログラムと組み合わせて開発する、といったことも可能です。

なお、教育の現場でも、学習すべきプログラミング言語として採用が増えつつあります。

▪ 多くのプログラミングパラダイムに対応している

プログラムには、その根本となる思想・発想法・開発法を表す各種の**パラダイム**があります。

Python は、**命令型プログラミング**、**手続き型プログラミング**、**関数型プログラミング**、**オブジェクト指向プログラミング**といった、複数の**プログラミングパラダイム**に対応します。すなわち、極めて、懐（ふところ）が深い言語です。

Python を学習すれば、上記のプログラミングパラダイムに精通できます。また、作成者が得意とするプログラミング技術や、作成するプログラムの性格などに応じて、採用するプログラミングパラダイム（や、その比率）を自由に変えることができます。

スクリプト言語である

スクリプト言語とは、プログラムの作成・実行・テストなどを行いやすい、比較的小規模な体系をもつプログラミング言語です。Pythonには、次のような特徴があります。

・**記述性が高い**。他の言語よりも数割ほど短く記述できる。
・**可読性が高い**。プログラムが読みやすい。
・**インタプリタ形式である**。プログラムを対話的に1行ずつ実行でき、試行錯誤しやすい。

ライブラリが豊富である

多くのプログラミング言語がそうなのですが、言語自体で行えることは限られています。グラフィック、ネットワークといった処理は、**ライブラリ**（処理を行うための**部品**が集められたもの）に任せます。

Pythonは、急速な普及とともに、幅広い分野のライブラリが充実しています。

プログラムの実行が高速ではない

インタプリタ形式であること（さらに、**動的な型付け言語**であることなど、いくつかの理由から）Pythonのプログラムは、決して高速には動作しません。

ただし、「高速な技術計算のライブラリを利用する」といった感じで、処理の主要部分を高速なライブラリなどに委ねることで、それなりの速度でプログラムを実行できます。

習得は決して容易ではない

Pythonは習得が容易、と宣伝されますが、そうではありません。記述性が高い、ということは、短い記述の中に、多くの深い意図が潜んでいるということです。また、Pythonには、いわゆる（習得が難しいといわれる）「ポインタ」は表面的にはありませんが、その内部はポインタ（参照）だらけです。学習においては、一つ一つの式や文の意味を正しく理解していく必要があります。

Pythonのバージョンについて

Pythonはバージョンアップを重ね続けています。そのバージョン番号は、A.BあるいはA.B.C形式で表されています。Aはメジャーバージョン番号、Bはマイナーバージョン番号です。末尾のCは、小変更やバグフィックスのときに上げられます。

1991年に0.9、1994年に1.0、2000年に2.0、2008年に3.0が発表されました。

Pythonが大きな注目を浴びたのは、２系からであり、数多くのライブラリが作成されました。３系は、言語レベルで大きな変化が施されました。２系と３系は、互換性に欠けており、２系で作成したプログラムの大部分は、そのままでは３系では動作しません。２系のサポートは2020年までです。２系用のライブラリを使用しなければならない、などの特別な理由がない限り、３系を使うべきです。

本書で学習するのは、Python 3.7です。

Pythonプログラムの実行

Pythonでプログラムを作る前に、まずPythonをインストールする必要があります。付録（p.375〜）を読んでインストールを行いましょう。

なお、付録でも学習しますが、Pythonのプログラムを実行する方法としては、主として3種類があります（**Fig.1-2**）。

インタラクティブシェル（基本対話モード）

プログラムを1行ずつ実行します（図a）。本章では、この方法のみを使います。

統合開発環境での実行

IDLE（*Integrated DeveLopment Environment*）と呼ばれる統合開発環境（図b）を使って実行します。

pythonコマンドによる実行

`python`コマンドに対して、（`'.py'`という拡張子を付けて）保存ずみのプログラムを与えて、実行します。

a インタラクティブシェル（基本対話モード）

```
Python 3.7 (64-bit)
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD6
4)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> |
Ln: 3 Col: 4
```

b 統合開発環境（IDLE）

Fig.1-2 Pythonプログラムの実行

なお、サードパーティーから提供される統合開発環境を使う方法や、コンパイルと呼ばれる作業を行うことで、高速にプログラムを実行する方法などもあります。

1-2 Pythonの基本

本節では、インタラクティブシェル（基本対話モード）を利用して、Pythonに慣れながら、基本的なことがらを学習します。

インタラクティブシェル（基本対話モード）

Pythonプログラムの実行方法に、いくつかの種類があることが分かりました。さっそく、基本対話モードとも呼ばれるインタラクティブシェルを使っていきましょう。

インタラクティブシェル（基本対話モード）の起動と終了

まずは、**python**コマンドによって、インタラクティブシェルを起動します。

▶ 起動の方法は、OSのバージョンやPythonのバージョンなどに依存します。以下に示すのは、あくまでも一例です。

- MS-Windowsでは、Powershellあるいはコマンドプロンプト上で**python**と入力します。なお、スタートメニューから、次のようにたどっていくことによっても起動できます。
  ［スタートメニュー］ ─ ［Python 3.7］ ─ ［Python 3.7 (64bit)］
- Linuxでは、シェルのプロンプトで**python**と入力します。
- Macでは、ターミナルで**python3**と入力します。

起動すると、右向きの不等号が3個並んだ**>>>**という基本プロンプト（*primary prompt*）が表示されます。

▶ >>>の後ろに1個のスペースが表示されます。

プロンプトの後ろには、いろいろなコマンドが入力できます。

まずは、**copyright**と入力してみましょう。そうすると、著作権情報が表示されます。

例1-1　インタラクティブシェル（基本対話モード）における著作権情報の表示

```
Python 3.7.0 on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> copyright⏎
Copyright (c) 2001-2018 Python Software Foundation.
All Rights Reserved.

Copyright (c) 2000 BeOpen.com.
All Rights Reserved.

Copyright (c) 1995-2001 Corporation for National Research Initiatives.
All Rights Reserved.

Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
>>>
```

▶ 赤文字が、みなさんが打ち込む箇所です。また、青文字は、Pythonのシェルによって表示される箇所です。なお、表示される内容は、Pythonのバージョンなどによって異なります。

インタラクティブシェルを本格的に使っていく前に、次のことを理解しておきましょう。

▪ 終了方法

いくつかの終了方法が用意されています。

▫ quit 関数／ exit 関数による終了

quit() あるいは **exit()** と入力します。まずは、**quit()** を試しましょう。

例 1-2　インタラクティブシェルの終了（その１：quit関数）

```
>>> quit()⏎
```

インタラクティブシェルが終了しました。もう一度起動して、**exit()** を打ち込みましょう。

例 1-3　インタラクティブシェルの終了（その２：exit関数）

```
>>> exit()⏎
```

さて、関数という用語が出てきました。関数や、() の意味などについては、今後少しずつ学習していきます。

▫ キー操作による強制的な終了

強制的に終了する場合は、以下のキー操作を行います。

- MS-Windows　　：［Control］キーを押しながら［Z］キーを押して、それから［Enter］キーを押す。
- Mac や Linux など：［Control］キーを押しながら［D］キーを押して、それから［Enter］キーを押す。

この方法は、実行中のプログラムが終了しなくなったときなどに行う、最終手段です。

▶［Control］＋［Z］や、［Control］＋［D］は、**ファイル終端文字**と呼ばれます。

▪ コマンドの履歴を呼び出す

打ち込みずみのコマンド（文や式など）と、同一あるいは類似したコマンドを打ち込むときに、最初から打ち込み直す必要はありません。

［↑］、［↓］、［Home］、［End］、［Page Up］、［Page Down］の各キーを使用すると、それまでに入力したコマンドが順に出てきます。

過去に打ち込んだものと同じコマンドを打ち込むのであれば、そのまま［Enter］キーを押します。また、少しだけ書きかえるのであれば、［←］と［→］でカーソルを移動して必要な修正や変更などを行った上で、［Enter］キーを押します。

重要　同一あるいは類似したコマンドを打ち込む際は、過去に打ち込みずみのものを取り出した上で修正・変更する。

演算子とオペランド

インタラクティブシェルを“電卓代わり”に使って、Python に慣れていきましょう。

まずは《四則演算》です。計算式を打ち込むと、演算結果が表示されます。

例 1-4　四則演算とべき乗

```
>>> 7 + 3⏎          ← 加算
10
>>> 7 - 3⏎          ← 減算
4
>>> 7 * 3⏎          ← 乗算
21
>>> 7 / 3⏎          ← 除算
2.3333333333333335
>>> 7 // 3⏎         ← 切捨て除算（除算結果の小数部を切り捨てる）
2
>>> 7 % 3⏎          ← 剰余（7を3で割った剰余）
1
>>> 7 ** 3⏎         ← べき乗（7の3乗）
343
>>> 7 * (3 + 2) * 4⏎ ← ()の中が先に計算される
140
```

注意：同一または類似した式が必要なときは、打ち込み直すのではなく、打ち込みずみの履歴を取り出した上で変更するとよい（方法は前ページ）。

実数としての 7÷3 を求めるのが 7 / 3

整数としての 7÷3 は《2 あまり 1》
商　の 2 を求めるのが // 演算子
剰余の 1 を求めるのが % 演算子

▶ ←以降の緑文字は、補足解説です（打ち込んだり表示されたりするわけではありません）。
式を打ち込むだけで、演算結果が表示される理由などは、p.18 で学習します。

プロンプトの直後（7 よりも前）にスペースを打ち込んではいけません。ただし、7 と + のあいだ、+ と 3 のあいだ、3 と ⏎ のあいだはスペースがあっても（なくても）構いません。

▶ その理由は、3–3 節で学習します。また、7 / 3 の演算結果の最後の桁が 3 ではなくて 5 となる理由は、第 5 章で学習します。

演算子とオペランド

演算を行うための + や - などの記号は**演算子**（えんざんし）（*operator*）と呼ばれ、7 や 3 といった演算の対象は**オペランド**（*operand*）と呼ばれます（**Fig.1-3**）。

加減算の演算子 + と - は、日常の計算と同じ記号ですが、乗除算の演算子は違います。乗算は×ではなく *、除算は÷ではなく /、演算結果の小数部を切り捨てる除算は //、剰余（あまり）を求める演算は % です（**Table 1-1**）。

また、単純な四則演算ではないのですが、《べき乗（じょう）》を求める演算子が ** です。

▶ 7 ** 3 は、7 * 7 * 7 を求めます。なお、二つの * のあいだにスペースを入れてはいけません。

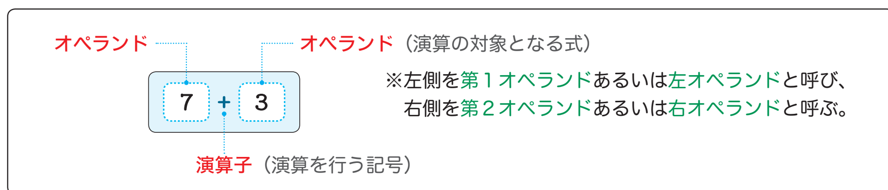


Fig.1-3　演算子とオペランド

最後に計算している 7 * (3 + 2) * 4 の ( ) が欠如すると、7 * 3 と 2 * 4 が加算され、演算結果は 29 となります。すなわち、以下の点も日常の算術演算と同じです。

- **四則演算は左から右へと行われる。**
- **乗除算は、加減算よりも優先される（演算子の優先度が高い）。**
- **先に行うべき演算は ( ) で囲む。**

( ) は入れ子(いれこ)にできます。たとえば、7 * ((3 + 5) % 2) といった具合です。なお、入れ子にすることを、“ネストする”といいます。

▶ 演算が左側から順に行われる演算子は、左結合です（べき乗 ** は例外的に右結合です）。演算子の結合規則については、p.75 で学習します。

演算子の優先度

利用した演算子は、オペランドが２個の**２項演算子**（*binary operator*）でした。２項演算子のほかに、オペランドが３個の**３項演算子**（*trinary operator*）と、オペランドが１個の**単項演算子**（*unary operator*）があります。

それでは、日常の計算でもおなじみの単項演算子 + と - を使ってみましょう。

例 1-5　２項演算子と単項演算子

```
>>> 7*+3↵                ← 7 * (+3)のこと：7と(+3)の積を求める
21
>>> 7*-3↵                ← 7 * (-3)のこと：7と(-3)の積を求める
-21
```

この例から、次のことが分かります。

- **単項 + 演算子と単項 - 演算子は、乗除演算子よりも優先して演算が行われる。**

＊

インタラクティブシェルを電卓代わりに使いながら、９個の演算子を学習しました。

これらの演算子は、優先度としては４種類に分かれます。Table 1-1 に示す一覧では、優先度が高いほうから順に並んでいます（優先度ごとに色分けしています）。

Table 1-1　基本的な算術演算子

x ** y	べき乗演算子	x を y 乗した値を生成。	右結合演算子
+x	単項 + 演算子	x そのものの値を生成。	
-x	単項 - 演算子	x の符号を反転した値を生成。	
x * y	乗算演算子	x に y を乗じた値を生成。	
x / y	除算演算子	x を y で除した値を生成（演算は実数で行われる）。	
x // y	切捨て除算演算子	x を y で除した値を生成（小数部を切り捨てて整数値を生成）。	
x % y	剰余演算子	x を y で除したときの剰余（あまり）を生成。	
x + y	加算演算子	x に y を加えた値を生成。	
x - y	減算演算子	x から y を減じた値を生成。	

数値型と数値リテラル

例 1-4 では、整数どうしの算術演算を行いました。演算子 / による演算結果だけが、小数部をもつ実数となっており、他の演算結果は、すべて整数です。

数値型

数値を表す方法は、プログラミング言語によって異なります。

数値や文字などを表す種類や方法のことを**型**（かた）（*type*）と呼ぶのですが、Python の数値を表す型には、3種類の**数値型**があります。

- **int 型**（イント）　整数を表す**整数型**（*integer type*）です。
- **float 型**（フロート）　実数を表す**浮動小数点数型**（*floating type*）です。
- **complex 型**（コンプレックス）　複素数を表す**複素数型**（*complex type*）です。

▶ 多くのプログラミング言語では、int 型で表せる数値は有限です。たとえば、「-2,147,483,648 から 2,147,483,647 までの範囲に収まらねばならない」といった制限がありますが、Python には、そのような値の制限はありません。
　また、float 型は、C 言語や Java 言語の（float 型ではなく）double 型に相当します（Python には double 型はありません）。

整数型と浮動小数点数型が混在した演算を行ってみましょう。

例 1-6　整数と浮動小数点数の演算

```
>>> 7 + 3        ← int   + int   の演算結果はint
10               この結果のみ int 型
>>> 7.0 + 3      ← float + int   の演算結果はfloat
10.0
>>> 7 + 3.0      ← int   + float の演算結果はfloat
10.0
>>> 7.0 + 3.0    ← float + float の演算結果はfloat
10.0
```

int 型どうしの加算結果は int 型ですが、それ以外の加算結果は float 型です。このように、オペランドの型によって、演算結果の型も変わります。詳細な規則は、第 5 章で学習します。

数値リテラル（整数リテラルと浮動小数点数リテラル）

さて、7 とか 3.0 といった数値を表す表記のことを**数値リテラル**（*numeric literal*）といいます。7 は**整数リテラル**（*integer literal*）で、3.0 は**浮動小数点数リテラル**（*floating point literal*）です。

▶ リテラルとは、『文字どおりの』『文字で表された』という意味です。

整数リテラルは、10 進数だけでなく、2 進数、8 進数、16 進数でも表記できます。

▶ 一般に、*n* 進数は、*n* を基数とする数です（**Column 1-2**：p.24）。

各基数のリテラルの表記は、次のとおりです。10 進以外は“前置き”が必要です。

- **2 進リテラル** … 前置きは 0b。使える数字は 0 ～ 1 の 2 個。
- **8 進リテラル** … 前置きは 0o。使える数字は 0 ～ 7 の 8 個。
- **10 進リテラル** … 使える数字は 0 ～ 9 の 10 個。
- **16 進リテラル** … 前置きは 0x。使える数字は 0 ～ 9 と a ～ f （A ～ F でも可）の 16 個。

▶ 2 進、8 進、16 進リテラルの先頭は、b と o と x を大文字にした 0B、0O、0X でも OK です（ただし、8 進数の 0O が読みづらくなります）。

0b や 0x で使われる 0 は、数値リテラルの前置き用の文字なので、値が 0 でない 10 進リテラルの先頭を 0 とすることはできません。すなわち、03 や 010 などは許されません。

整数リテラルを打ち込んで、確かめましょう。

例 1-7　2 進／8 進／10 進／16 進リテラル

```
>>> 0b10⏎          ← 2進数の10（10進数の2）
2
>>> 0o10⏎          ← 8進数の10（10進数の8）
8
>>> 10⏎            ← 10進数の10
10
>>> 0x10⏎          ← 16進数の10（10進数の16）
16
>>> 010⏎           ← 10進数の10とはみなされない
  File "<stdin>", line 1
    010
      ^
SyntaxError: invalid token
```

10 進リテラルの前に 0 を置くとエラーになる

最後の 010 では、エラーが発生しました。

▶ ここで発生しているのは、《構文エラー》です。第 3 章で学習します。

この例から、次のことが分かります。

- **演算を行わずに、数値リテラルのみを打ち込んでも、その値が表示される。**
- **整数値は 10 進数で表示される。**

Python 3.6 からは、数値リテラルの途中の好きな箇所に下線 _ を入れられます（実質的に無視されます）。桁数の多い数値を読みやすく表記できます。

例 1-8　整数リテラルに下線を含める

```
>>> 38_239_521_489_247⏎     ← 10進数の38,239,521,489,247
38239521489247
```

浮動小数点数リテラルは、整数部と小数部の一方を省略できます。また、10 の指数表記を末尾に付加できます。以下に示すのが、一例です。

```
6.52    10.    .001    1e5    3.14e-7    3.141_592_653_5
```

▶ e5 は 10^5 を表し、e-7 は 10^{-7} を表します。

文字列リテラルとエスケープシーケンス

ここまでは《数値》を扱ってきました。次は、《文字》を扱うことにします。早速、打ち込んで試してみましょう。

例 1-9　文字の並び（エラー）

```
>>> Fukuoka⏎
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Fukuoka' is not defined
```

単なる文字の並びは文字列ではない

残念ながら、エラーが発生しました。というのも、ここで打ち込んだ Fukuoka は、文字の並びではなく、《名前》として認識されるからです。

▶ エラーメッセージの最後の行を直訳すると、次のようになります。
名前エラー：'Fukuoka' という名前は定義されていません。

その《名前》が、何の名前なのかは後回しにして、文字の学習を進めましょう。

文字列リテラル

文字の並びは**文字列**（*string*）と呼ばれ、その綴（つづ）りを表記するのが**文字列リテラル**（*string literal*）です。

さて、その文字列リテラルは、表したい文字の並びを、単一引用符 ' で囲んで、'A' や 'Fukuoka' と表記します。それでは、確かめましょう。

例 1-10　文字列リテラルと加算／乗算

```
>>> 'A'⏎                         ← 1個の文字で構成される文字列
'A'
>>> 'Fukuoka'⏎                   ← 7個の文字で構成される文字列
'Fukuoka'
>>> '福岡'⏎                      ← 2個の文字で構成される文字列
'福岡'
>>> '福' + 岡' + '市'⏎           ← 演算子+によって文字列を連結
'福岡市'
>>> '福岡' * 3⏎                  ← 演算子*によって文字列を繰り返す（3回）
'福岡福岡福岡'
>>> 3 * '福岡' + 'です。'⏎       ← 演算子*による繰返しと演算子+による連結
'福岡福岡福岡です。'
>>> 3 * '福岡' * 2⏎              ← 演算子*によって文字列を繰り返す（6回）
'福岡福岡福岡福岡福岡福岡'
>>> ''⏎                          ← 空の文字列
''
```

これも立派な文字列

後半の対話例から、以下のことが分かります。

- “文字列 + 文字列” の演算によって、左側から連結した文字列が得られる。
- 文字列と整数を * 演算子で掛けあわせると、その回数だけ繰り返された文字列が得られる。
  ※注意：文字列どうしを * 演算子で掛けあわせることはできない。
- 文字列を構成する文字は 0 個でもよい（空の文字列）。

数値と文字列は、性質がまったく異なります。文字列リテラルの型は、int 型や float 型ではなく、文字列を表す **str 型**です。

文字列と数値の違いを、以下の例で確認しましょう。

例 1-11　文字列どうしの加算と整数との加算

```
>>> '12' + '34'⏎          ⬅ 文字列の加算（連結）：46ではない
'1234'
>>> 'Python' + 3⏎         ⬅ 'Python3'ではない（'Python' + '3'であれば'Python3'）
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

“文字列 + 数値”の演算は行えないことが分かります。

エスケープシーケンス

単一引用符 ' は、文字列リテラルの最初と最後を表すための、特別な文字です。そのため、たとえば 'This isn't a pen.' といった表記は行えません。

文字 ' は、２個の文字 \' で表記します（見かけは \ と ' の２個の文字ですが、' という１個の文字を表します）。すなわち、'This isn\'t a pen.' が正しい表記です。

× 'This isn't a pen.'　　　○ 'This isn\'t a pen.'

このような、逆斜線記号 \ を先頭にした、２個、あるいは、それ以上の個数の文字で、（通常の文字としては表記不能あるいは困難である）単一の文字を表す表記法が、**エスケープシーケンス**（*escape sequence*）です。

Table 1-2 に示すのが、その一覧です。

Table 1-2　エスケープシーケンス

\a	警報（*alert*）	聴覚的または視覚的な警報を発する。
\b	後退（*backspace*）	表示位置を直前の位置へ移動する。
\f	書式送り（*form feed*）	改ページして、次のページの先頭へ移動する。
\n	改行（*new line*）	改行して、次の行の先頭へ移動する。
\r	復帰（*carriage return*）	現在の行の先頭位置へ移動する。
\t	水平タブ（*horizontal tab*）	次の水平タブ位置へ移動する。
\v	垂直タブ（*vertical tab*）	次の垂直タブ位置へ移動する。
\\	逆斜線文字 \	
\?	疑問符 ?	
\'	単一引用符 '	
\"	二重引用符 "	
\newline	バックスラッシュと改行文字を無視する。	p.21
\ooo	ooo は１～３桁の８進数	８進数で ooo の値をもつ文字。
\xhh	hh は2桁の 16 進数	16 進数で hh の値をもつ文字。

▶ 注意：日本語版の MS-Windows では、逆斜線 \ の代わりに円記号 ¥ を使います（p.20）。

文字列リテラルの表記方法

単一引用符 ' 1個で構成される文字列リテラルは、文字列リテラル開始の ' と、単一引用符を表す \' と、文字列リテラル終了の ' を合わせた4文字です。確かめましょう。

例 1-12　単一引用符

```
>>> '\''↵          ← 単一引用符1個の文字のみで構成される文字列
"'"
```

表示された結果が、これまでと少々違います。文字列を囲む記号が、単一引用符 ' ではなく、二重引用符 " になっています。

実は、文字列リテラルの表記方法には、以下の4種類があります。

- **単一引用符 ' で囲む**

  二重引用符 " をそのまま表記できる。単一引用符は \' で表記する。

- **二重引用符 " で囲む**

  単一引用符 ' をそのまま表記できる。二重引用符は \" で表記する。

- **3個の単一引用符 ''' で囲む**

  二重引用符 " をそのまま表記でき、途中に改行文字を含めることができる。

- **3個の二重引用符 """ で囲む**

  単一引用符 ' をそのまま表記でき、途中に改行文字を含めることができる。

引用符や改行文字を含まない限り、どれも同じように使えます。確認しましょう。

例 1-13　4種類の文字列リテラル

```
>>> 'String'↵
'String'
>>> "String"↵
'String'
>>> '''String'''↵
'String'
>>> """String"""↵
'String'
```

文字列を囲む記号は4種類

インタラクティブシェルが表示する結果は、すべて単一引用符 ' で囲まれており、すべての場合で 'String' と表示されます。

それでは、文字列リテラル内に引用符や改行を入れてみましょう。

例 1-14　引用符／改行を含む文字列リテラル

```
>>> 'それは"ABC"です。'↵
'それは"ABC"です。'
>>> "文字列\"ABC\"を構成するのは'A'と'B'と'C'です。"↵
'文字列"ABC"を構成するのは\'A\'と\'B\'と\'C\'です。'
>>> '''途中で↵
... 改行↵
... できます。'''↵
'途中で\n改行\nできます。'
```

注意：''' あるいは """ の記述の途中で改行を行うと、基本プロンプト >>> ではなく、補助プロンプト（secondary prompt）と呼ばれる ... が表示される。

さて、これまでの表示結果から、インタラクティブシェルは、文字列リテラルの内容を以下のように表示することが分かります。

- 基本的には '' で囲んで表示する。
- 文字列内に単一引用符が含まれていれば "" で囲んで表示する。
- 改行文字と単一引用符は、エスケープシーケンス \n と \' として表示する。

複数の記述ができるようになっているのは、記述の際に便利だからです。みなさんがプログラムを書くときは、なるべく統一するとよいでしょう。

▶ 本書では、基本的に '' で囲み、単一引用符が含まれているときに "" で囲みます。

さて、''' 形式と """ 形式は、改行文字を含む文字列リテラルを表すとき以外は、使い道がないように感じられるかもしれませんが、そうではありません。

プログラム中に埋め込んだ記述をもとに、プログラムのドキュメント（マニュアルのようなもの）を生成する機能があります。""" 形式の文字列リテラルは、その記述で使います。

▶ """ 形式の文字列リテラルについては、第９章で学習します。

隣接した文字列リテラルの結合

スペース、タブ、改行などの空白文字をはさんで並べられた文字列リテラルは、連続して記述されたものとみなされます。

たとえば、'ABC'　'DEF' は結合されて、'ABCDEF' とみなされます。確認しましょう。

例 1-15　空白をはさんで隣接した文字列リテラル（字句上の結合）

```
>>> 'ABC'  'DEF'⏎
'ABCDEF'
```

空白が除去されて結合された 'ABCDEF' が表示されます（すなわち、'ABC' + 'DEF' のように演算によって連結されるのではありません）。

原文字列リテラル

r もしくは R の“前置き”付きの文字列リテラルは、**原文字列リテラル**（*raw string literal*）となります。原文字列リテラルでは、その中に置かれたエスケープシーケンスが、その綴りどおりに解釈されます。

以下に示すのが、文字列リテラルと原文字列リテラルの表記の違いの具体例です。

逆斜線文字 \ が４個連続する文字列リテラル

文字列リテラル	'\\\\\\\\'	※エスケープシーケンス \\ が４個
原文字列リテラル	r'\\\\'	※逆斜線文字 \ が４個

逆斜線が頻出する“パス”の表記などで使います。

▶ raw は、『生（なま）の』『未加工の』という意味であり、原文字列リテラルは、『生文字列リテラル』とも呼ばれます。

変数と型

引用符で囲まれていない **Fukuoka** は、文字列リテラルではなくて名前でした（p.12）。

何の名前かというと、ずばり**変数**（*variable*）の名前です。変数については、次章以降で詳しく学習しますので、とりあえずは、次のように理解しておきます。

> **重要** **変数**とは、整数や浮動小数点数や文字列などの**値**を格納するための《箱》のようなものであり、いったん値を入れておけば、いつでも取り出せる。

▶ なお、この理解は不正確であって、のちのち完全に覆（くつがえ）されます（主に第 5 章で学習します）。

それでは、実際に確かめましょう。

例 1-16　変数（値の代入と演算）

```
1 >>> x = 17↵            ← xに17を代入する
  >>> y = 52↵            ← yに52を代入する
  >>> z = x + y↵         ← zにx + yを代入する
2 >>> x↵                 ← xの値を評価
  17
  >>> y↵                 ← yの値を評価
  52
  >>> z↵                 ← zの値を評価
  69
3 >>> x + 2↵             ← x + 2の値を評価
  19
  >>> x // 2↵            ← x // 2の値を評価
  8
4 >>> x, y, z↵           ← xとyとzの値を評価
  (17, 52, 69)
```

評価という用語については、第3章で詳しく学習します（p.54）。

1 **=** は代入を指示する記号です。数学のように“x と 17 が等しい”といっているのではありません。ここでは、変数 **x** と **y** と **z** に値を代入しています。

2 インタラクティブシェル上で変数名だけを打ち込むと、その変数の値が表示されます。

3 算術演算を行うと、その演算結果が表示されます。

4 コンマ **,** で区切って複数の式を打ち込むと、対応する各値をコンマで区切ったものが **( )** で囲まれて表示されます。これは、便利な方法です。

例 **1-9**（p.12）では、**Fukuoka** という名前を打ち込むと、その名前が定義されていないことによるエラーが発生しました。しかし、今回のように、“**x = 17**” という代入であれば、エラーは発生しません。そうなるのは、以下の理由によります：

> **重要** 初めて使う名前の変数に値を**代入**すると、その名前の変数が自動的に用意される。

三つの変数 **x** と **y** と **z** は、整数値が代入されているため、**int** 型として用意されます。

ところが、変数の型は固定されておらず、変更できるようになっています。以下の例を打ち込んで、確認しましょう。

例 1-17　変数の型を変化させる

変数の型を変化させて《値》を確認

```
>>> x = 17↵           ← xにint型の整数値17を代入
>>> x↵                ← xの値を評価
17
>>> x = 3.14↵         ← xにfloat型の浮動小数点数値3.14を代入
>>> x↵                ← xの値を評価
3.14
>>> x = 'ABC'↵        ← xにstr型の文字列'ABC'を代入
>>> x↵                ← xの値を評価
'ABC'
```

変数 x が、整数の int 型、浮動小数点数の float 型、文字列の str 型と、次々と《変身》しています。

type 関数による型の調査

実は、型を確認する方法があります。type(式) とすると、( ) で囲まれた式の型が得られます。確認しましょう。

例 1-18　変数の型の確認

変数の型を変化させて《型》を確認

```
>>> x = 17↵
>>> type(x)↵
<class 'int'>
>>> x = 3.14↵
>>> type(x)↵
<class 'float'>
>>> x = 'ABC'↵
>>> type(x)↵
<class 'str'>
```

変数 x の型が、次々と変身していく様子が確認できました。

なお、type() は、リテラルに対しても適用できます。確かめましょう。

例 1-19　リテラルの型の確認

リテラルの《型》を確認

```
>>> type(5)↵
<class 'int'>
>>> type(5.5)↵
<class 'float'>
>>> type('ABC')↵
<class 'str'>
```

ここまでの変数名は、単純な 1 文字の名前でした。どんな名前でも与えられるわけではありません。大雑把な規則は次のとおりです。

- 使える文字は、アルファベットと数字と下線。
- アルファベットの大文字と小文字は区別される。
- 数字を先頭文字にすることはできない。

たとえば、次のような名前が利用できます。

a　abc　point　point_3d　a1　x2

▶ 変数名には漢字も使えますが、お薦めできません。詳細な規則は、p.75 で学習します。

式と文

代入と加算を比べましょう。以下のように打ち込んでみます。

例 1-20　代入と加算

```
>>> x = 17⏎          ← xに17を代入する
>>> x + 17⏎          ← xに17を加えた値
34
```

最初の“x = 17”では何も表示されませんが、続く“x + 17”では値（演算の結果）が表示されます。次のような決定的な違いがあるからです。

- x = 17 は、文（*statement*）である。　※ 式ではない文である。
- x + 17 は、式（*expression*）である。　※ ただし文でもある（p.52）。

式を打ち込むと、その値が表示されるのですが、文を打ち込んでも、その処理（この例では代入）が行われるだけです。

▶ もちろん、表示を命じる文（次章で学習します）を実行すると、表示が行われます。

“x + 17”が式であって、“x = 17”が式でないことの確認は容易です：

例 1-21　代入と加算の型

```
>>> x = 0⏎
>>> type(x + 17)⏎
<class 'int'>                                  ← 加算 x + 17 は int 型
>>> type(x = 17)⏎
Traceback (most recent call last):             ← 代入 x = 17 には型がない
  File "<stdin>", line 1, in <module>
TypeError: type() takes 1 or 3 arguments
```

式ではない後者は、《型》を調べられないめ、エラーとなります。式と文という言葉は、いずれも漢字で1文字ですが、奥深いものです。次章以降で少しずつ学習していきます。

代入文

既に学習したとおり、+ は加算の演算子です。その一方で、= は演算子でありません。

重要　代入を行うための記号 = は、演算子ではない。

▶ Table 3-5（p.76）に示している全演算子の一覧に = は含まれていません。ただし、演算子ではないにもかかわらず、（慣習上）代入演算子と呼ばれますので、本書もそれにしたがっています。

記号 = を使って代入を行う文は**代入文**（*assignment statement*）と呼ばれます。代入文は、極めて（おそらく、みなさんの想像をはるかに超えるくらい）多機能です。

▶ 初めて使う名前の変数に値を代入するだけで、変数が自動的に用意される機能を、学習ずみです。

詳細は、少しずつ学習していきます。ここでは、二つの便利な機能を学習します。

複数の変数への同一値の一括代入を行う

複数の変数に対して、同一の値を一括して代入できます。確かめましょう。

例 1-22　複数の変数への同一値の代入

```
>>> x = y = 1↵              ← xとyの両方に1を代入する
>>> x↵
1
>>> y↵
1
```

x と y の二つの変数（が自動的に用意されるとともに）に 1 が代入されます。

複数の変数への異なる値の一括代入を行う

複数の変数に対する代入をまとめて行えます。確かめましょう。

例 1-23　複数の変数への異なる値の一括代入

```
>>> x, y, z = 1, 2, 3↵      ← xとyとzに、それぞれ1と2と3を代入する
>>> x↵
1
>>> y↵
2
>>> z↵
3
```

変数 x と y と z に、それぞれ 1 と 2 と 3 が代入されます。

それでは、ちょっとした応用例を試してみましょう。

例 1-24　複数の変数への一括代入を同時に行う

```
>>> x = 6↵
>>> y = 2↵
>>> x, y = y + 2, x + 3↵    ← xにy+2を代入／yにx+3を代入
>>> x↵
4
>>> y↵
9
```

x への代入と、y への代入が指示されています。これらの代入が、逐次（ちくじ）（順番に）行われるのであれば、以下のようになるはずです。

① x = y + 2 によって、x は 4 に更新される。
② y = x + 3 によって、（更新された x の 4 と 3 の和が代入されて）y は 7 になる。　×

実際は、そうではありません。二つの代入は（論理的に）同時に行われます。すなわち、

▫ x = y + 2 によって、x は 4 になる。
▫ y = x + 3 によって、y は 9 になる。　○

となります。

重要　複数の変数に対する一括代入は、論理的に同時に行われる。

▶ コンマを使った複数の変数への代入には、第 8 章で学習する**タプル**が使われています。

記号文字の読み方

Python で利用する記号文字の読み方を、俗称（ぞくしょう）を含めてまとめたのが **Table 1-3** です。

Table 1-3　記号文字の読み方

記号	読み方
+	プラス符号、正符号、プラス、たす
-	マイナス符号、負符号、ハイフン、マイナス、ひく
*	アステリスク、アスタリスク、アスター、かけ、こめ、ほし
/	スラッシュ、スラ、わる
\	逆斜線、バックスラッシュ、バックスラ、バック　※JIS コードでは ¥
¥	円記号、円、円マーク
%	パーセント
.	ピリオド、小数点文字、ドット、てん
,	コンマ、カンマ
:	コロン、ダブルドット
;	セミコロン
'	単一引用符、一重引用符、引用符、シングルクォーテーション
"	二重引用符、ダブルクォーテーション
(	左括弧、開き括弧、左丸括弧、始め丸括弧、左小括弧、始め小括弧、左パーレン
)	右括弧、閉じ括弧、右丸括弧、終り丸括弧、右小括弧、終り小括弧、右パーレン
{	左波括弧、左中括弧、始め中括弧、左ブレイス、左カーリーブラケット、左カール
}	右波括弧、右中括弧、終り中括弧、右ブレイス、右カーリーブラケット、右カール
[	左角括弧、始め角括弧、左大括弧、始め大括弧、左ブラケット
]	右角括弧、終り角括弧、右大括弧、終り大括弧、右ブラケット
<	小なり、左アングル括弧、左向き不等号
>	大なり、右アングル括弧、右向き不等号
?	疑問符、はてな、クエッション、クエスチョン
!	感嘆符、エクスクラメーション、びっくりマーク、びっくり、ノット
&	アンド、アンパサンド
~	チルダ、チルド、なみ、にょろ　※JIS コードでは ‾（オーバライン）
‾	オーバライン、上線、アッパライン
^	アクサンシルコンフレックス、ハット、カレット、キャレット
#	番号記号、ナンバー、ハッシュ、スクエア、オクトソープ、ダブルクロス、井桁（いげた）
_	下線、アンダライン、アンダバー、アンダスコア
=	等号、イクオール、イコール
|	縦線、バーチカルライン

注意 !!

▶ 注意：日本語版の MS-Windows では、逆斜線 \ の代わりに円記号 ¥ を使います。たとえば、改行文字を表すエスケープシーケンス `\n` は、`¥n` となります。
お使いのシステムが ¥ を使う環境であれば、本書のすべての \ を ¥ と読みかえてください。

\ による行の継続

行の終端（改行文字の直前）に \ を置くと、現在の行が、そのまま次の行へと継続します。すなわち、\ と改行文字を連続記述したエスケープシーケンス（**Table 1-2**：p.13）は、

この行は、次の行に続きますよ。

という目印です。確かめましょう。

例 1-25　行末の\による行の継続

```
>>> x \↵                ← 途中で改行（次行に続く）
... = 17↵               ← 前の行の続き                 x = 17
>>> x↵
17
>>> x = 5    \↵         ← 途中で改行（次行に続く）      x = 5 + 3
... + 3 ↵               ← 前の行の続き
>>> x↵
8
>>> x = \ ↵             \の後ろにスペースを置くとエラーになる
  File "<stdin>", line 1
    x = \
         ^
SyntaxError: unexpected character after line continuation character
```

最後の例のように、\ と改行文字のあいだにスペースがあると、エラーとなります。

＊

\ による行の継続は、インタラクティブシェル特有ではありません。次章以降で学習する、スクリプトプログラム内でも利用できます。

重要 現在の行の続きを次行にもちこしたいときは、行末に \ を置く。
※行の終端の文字が \ となっている行は、次の行に続く。

▶ この他にも、カッコ記号 ( ) による継続などもあります（第 3 章で学習します）。

Column 1-1　インタラクティブシェルで最後に表示した値

インタラクティブシェルで最後に表示した値は、下線 1 個の _ という名前の変数に入れられます。この変数は、計算結果をもとに次の計算を行う際に有用です。

以下に示すのが、その利用例です。

```
>>> 5 + 3↵
8
>>> _ * 2↵              ← 最後に表示した値 * 2
16
>>> 7 + _↵              ← 7 + 最後に表示した値
23
```

▶ 電話のボタンでも使われていて、Python のプログラムでも多用される # は、**番号記号**であって、横線が右上がりに傾く音楽のシャープ記号♯とは、まったく異なる記号文字です。
ただし、# は、日本ではシャープと勘違いされたまま定着しています。

Pythonの哲学

インタラクティブシェルで、以下のように打ち込んでみましょう。

例 1-26 import thisの実行によるThe Zen of Pythonの表示

```
>>> import this⏎
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

ズラズラズラっと英語の文書が表示されます。これは、Tim Peters氏によってまとめられた"The Zen of Python"です。Zenは、日本語の《禅(ぜん)》に由来します。

▶ 同じ内容が、PEP内の下記のサイトに掲載されています。
https://www.python.org/dev/peps/pep-0020/
※PEPについては、p.82で学習します。

それでは、ひととおり読んでいきましょう(現時点では理解できなくても構いません)。

- **Beautiful is better than ugly.**

醜いよりも美しいほうがよい。

- **Explicit is better than implicit.**

暗示するより明示するほうがよい。

- **Simple is better than complex.**

複雑であるよりも単純なほうがよい。

- **Complex is better than complicated.**

複雑すぎるよりも、ただ複雑なほうがよい。

- **Flat is better than nested.**

ネストしているよりも、しないほうがよい。

- Sparse is better than dense.

  密よりも疎のほうがよい。

- Readability counts.

  可読性（読みやすさ）が大切だ。

- Special cases aren't special enough to break the rules.

  特殊だからといって規則を破る理由にならない。

- Although practicality beats purity.

  とはいえ、実用性は、純粋さに勝る。

- Errors should never pass silently.

  エラーを黙って渡してはならない。

- Unless explicitly silenced.

  とはいえ、わざと隠されているのならば見逃せばよい。

- In the face of ambiguity, refuse the temptation to guess.

  曖昧なものに出会ったときに、その意味を推測してはならない。

- There should be one-- and preferably only one --obvious way to do it.

  何かよい方法があるはずだ。誰にとっても明らかな、唯一の方法が。

- Although that way may not be obvious at first unless you're Dutch.

  オランダ人でない限り、そのような方法が明らかとは思えないだろうけど。

  ▶ “オランダ人”は、Python 開発者の Guido van Rossum 氏がオランダ人であることに由来します。

- Now is better than never.

  いつまでもやらないのではなく、やるのは今でしょ。

- Although never is often better than *right* now.

  とはいえ、今《すぐ》にやるより、やらないほうがよいことも多い。

- If the implementation is hard to explain, it's a bad idea.

  実装を説明するのが難しければ、アイディアが悪い。

- If the implementation is easy to explain, it may be a good idea.

  実装が説明しやすければ、アイディアがよいはずだ。

- Namespaces are one honking great idea -- let's do more of those!

  複数の名前空間の使い分けは、とても優れたアイディアだ。他にもたくさんアイディアを使おう。

Column 1-2 基数について

私たちが日常の計算で利用する10進数は、「10を**基数**とする数」です。

同様に、2進数は「2を基数とする数」であり、8進数は「8を基数とする数」であり、16進数は「16を基数とする数」です。

各基数について簡単に学習していきましょう。

▪10進数

10進数では、以下に示す10種類の数字で数を表現します。

```
0 1 2 3 4 5 6 7 8 9
```

これらを使い切ったら、桁が繰り上がって10となります。2桁の数は、10から始まって99までです。その次は、さらに繰り上がった100です。すなわち、以下のようになります。

1桁　…　0から9までの10種類の数を表す。
～2桁　…　0から99までの100種類の数を表す。
～3桁　…　0から999までの1,000種類の数を表す。

10進数の各桁は、下の桁から順に10^0, 10^1, 10^2, … と、10のべき乗の重みをもちます。そのため、たとえば1234は、次のように解釈できます。

$$1234 = 1\times10^3 + 2\times10^2 + 3\times10^1 + 4\times10^0$$

※10^0は1です（2^0でも8^0でも、とにかく0乗の値は1です）。

▪2進数

2進数では、以下に示す2種類の数字で数を表現します。

```
0 1
```

これらを使い切ったら、桁が繰り上がって10となります。2桁の数は、10から始まって11までです。その次は、さらに繰り上がった100です。すなわち、以下のようになります。

1桁　…　0から1までの2種類の数を表す。
～2桁　…　0から11までの4種類の数を表す。
～3桁　…　0から111までの8種類の数を表す。

2進数の各桁は、下の桁から順に2^0, 2^1, 2^2, … と、2のべき乗の重みをもちます。そのため、たとえば1011は、次のように解釈できます。

$$1011 = 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times1^0$$

10進数で表すと11です。

※Pythonの2進整数リテラルの表記では、`0b1011`となります。

▪8進数

8進数では、以下に示す8種類の数字で数を表現します。

```
0 1 2 3 4 5 6 7
```

これらを使い切ったら、桁が繰り上がって10となり、さらにその次の数は11となります。2桁の数は、10から始まって77までです。これで2桁を使い切りますので、その次は100です。すなわち、以下のようになります。

1 桁　　…　0 から 7 までの 8 種類の数を表す。
～ 2 桁　…　0 から 77 までの 64 種類の数を表す。
～ 3 桁　…　0 から 777 までの 512 種類の数を表す。

8 進数の各桁は、下の桁から順に 8^0, 8^1, 8^2, … と、8 のべき乗の重みをもちます。そのため、たとえば 5316 は、次のように解釈できます。

$$5316 = 5\times8^3 + 3\times8^2 + 1\times8^1 + 6\times8^0$$

10 進数で表すと 2766 です。
※ Python の 8 進整数リテラルの表記では、**0o5316** となります。

■ 16 進数

16 進数では、以下に示す 16 種類の数字で数を表現します。

0 1 2 3 4 5 6 7 8 9 A B C D E F

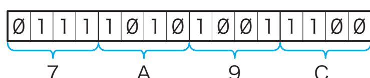
先頭から順に、10 進数の 0 ～ 15 に対応します（**A** ～ **F** は、小文字の **a** ～ **f** も使えます）。
これらを使い切ったら、桁が繰り上がって 10 となります。2 桁の数は、10 から始まって FF までです。その次は、さらに繰り上がった 100 です。
16 進数の各桁は、下の桁から順に 16^0, 16^1, 16^2, … と、16 のべき乗の重みをもちます。そのため、たとえば 12A3 は、次のように解釈できます。

$$12A3 = 1\times16^3 + 2\times16^2 + 10\times16^1 + 3\times16^0$$

10 進数で表すと 4771 です。
※ Python の 16 進整数リテラルの表記では、**0x12a3** となります。

■ 2進数と16 進数の相互変換

Table 1C-1 に示すように、4桁の2進数は、1桁の 16 進数に対応します（すなわち、4桁の2進数で表せる 0000 ～ 1111 は、1桁の 16 進数 0 ～ F です）。
このことを利用すると、2進数から 16 進数への基数変換、あるいは 16 進数から2進数への基数変換は、容易に行えます。
たとえば、2進数 0111101010011100 を 16 進数に変換するには、4桁ごとに区切って、それぞれを1桁の 16 進数に置きかえるだけです。



なお、16 進数から2進数への変換では、逆の作業を行います（16 進数の1桁を2進数の4桁に置きかえます）。
なお、8進数の1桁は、2進数の3桁に対応しています（3桁の2進数で表せる 000 ～ 111 は、1桁の8進数 0 ～ 7 です）。
このことを利用すると、同様な変換が行えます。

Table 1C-1　2進数と16進数の対応

2進数	16進数	2進数	16進数
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

まとめ

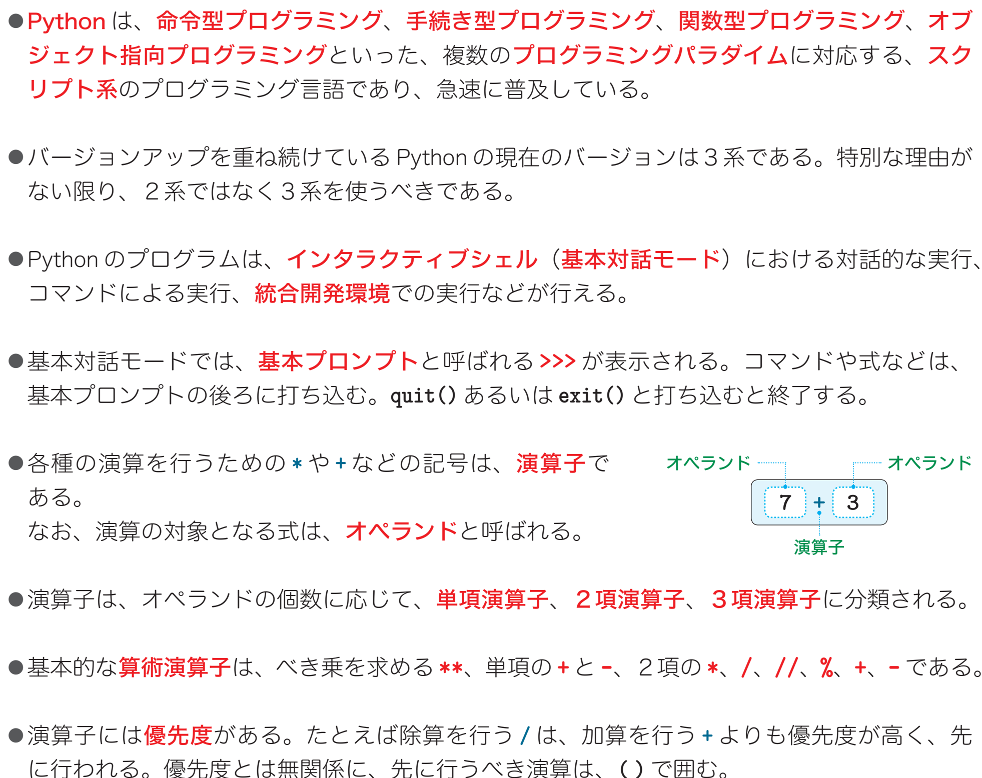
●Python は、**命令型プログラミング**、**手続き型プログラミング**、**関数型プログラミング**、**オブジェクト指向プログラミング**といった、複数の**プログラミングパラダイム**に対応する、**スクリプト系**のプログラミング言語であり、急速に普及している。

●バージョンアップを重ね続けている Python の現在のバージョンは３系である。特別な理由がない限り、２系ではなく３系を使うべきである。

●Python のプログラムは、**インタラクティブシェル**（**基本対話モード**）における対話的な実行、コマンドによる実行、**統合開発環境**での実行などが行える。

●基本対話モードでは、**基本プロンプト**と呼ばれる **>>>** が表示される。コマンドや式などは、基本プロンプトの後ろに打ち込む。`quit()` あるいは `exit()` と打ち込むと終了する。

●各種の演算を行うための * や + などの記号は、**演算子**である。
なお、演算の対象となる式は、**オペランド**と呼ばれる。

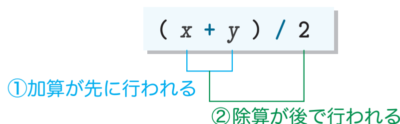


●演算子は、オペランドの個数に応じて、**単項演算子**、**２項演算子**、**３項演算子**に分類される。

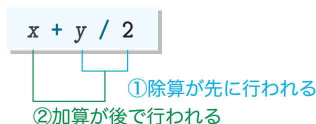
●基本的な**算術演算子**は、べき乗を求める **、単項の + と -、２項の *、/、//、%、+、- である。

●演算子には**優先度**がある。たとえば除算を行う / は、加算を行う + よりも優先度が高く、先に行われる。優先度とは無関係に、先に行うべき演算は、( ) で囲む。

xとyの平均を求める



xに $\frac{y}{2}$ を加える



●数値や文字の特性を表すのが**型**である。

●**数値型**には、**整数型**（`int` 型）、**浮動小数点数型**（`float` 型）、**複素数型**（`complex` 型）がある。

●数字や文字の並びによって、数値を表す表記が、**数値リテラル**である。

●**整数リテラル**は、**２進リテラル**、**８進リテラル**、**10 進リテラル**、**16 進リテラル**の４種類の基数で表記できる。

●**浮動小数点数リテラル**の末尾には、10 の指数表記を e^n 形式で付加できる。

● 日本語版の Microsoft Windows では、逆斜線記号 \ の代わりに円記号 ¥ を使う。

● 逆斜線記号 \ を先頭にした複数個の文字によって、単一の文字を表す**エスケープシーケンス**には、改行文字 `\n`、復帰文字 `\r` などがある。

● 行の終端（改行文字の直前）に \ を置くと、現在の行を、次の行へと継続できる。

● 文字の並びを表す型は**文字列型**（**str 型**）である。その綴りを表記する**文字列リテラル**は、表すべき文字の並びを、単一引用符 '、二重引用符 "、それらを 3 個並べた ''' もしくは """ で囲む。

● 空白文字をはさんで隣接した文字列リテラルは結合される。

● **原文字列リテラル**は、その中に含まれるエスケープシーケンスが、綴りどおりに解釈される。

● 演算子 + による数値どうしの加算、文字列どうしの加算（連結）は行えるが、文字列と数値の加算は行えない。

● **変数**には、整数や浮動小数点数や文字列などの**値**を格納でき、その値はいつでも取り出せる。

● `x + 17` は**式**であるが、`x = 17` は式ではなく**文**である。**代入**を行うための記号 = は、極めて多機能であって、演算子ではない。

● 初めて使う名前の変数に値を代入すると、その名前の変数が自動的に用意される。

● 変数には、現在の型とは異なる型の値を代入できる。

● 複数の変数をコンマ , で区切ったものを左辺に置けば、一度に複数の値を代入できる。それらの代入は、論理的には同時に行われる。

● 変数やリテラルの型は、**type(式)** で調べられる。

● Tim Peters 氏によってまとめられた、19 項目で構成される “**The Zen of Python**” は、世界中の Python プログラマに愛読されている、Python プログラミングの指針である。

`x ** y`	べき乗演算子	x を y 乗した値（右結合）	`7 ** 3` ➡ `343`
`+x`	単項 + 演算子	x そのものの値	`+7` ➡ `7`
`-x`	単項 - 演算子	x の符号を反転した値	`-7` ➡ `-7`
`x * y`	乗算演算子	x に y を乗じた値	`7 * 3` ➡ `21`
`x / y`	除算演算子	x を y で除した値（実数値）	`7 / 3` ➡ `2.3333333333333335`
`x // y`	切捨て除算演算子	x を y で除した値（小数部は切捨て）	`7 // 3` ➡ `2`
`x % y`	剰余演算子	x を y で除したときの剰余（あまり）	`7 % 3` ➡ `1`
`x + y`	加算演算子	x に y を加えた値	`7 + 3` ➡ `10`
`x - y`	減算演算子	x から y を減じた値	`7 - 3` ➡ `4`