Python3 入門

< 目次 >

1章 Pythonとは

1．1 Pythonの特徴

Python は Guido van Rossum 氏により「習得が容易で読みやすく効率のよいコードが簡潔に書ける」という設計思想で作られた、オブジェクト指向の軽量プログラミング言語（Lightweight Language）です。

「習得の容易さ」の一つとして、Python はオブジェクト指向言語なのですがクラスを使っていることを意識させない手続き型プログラミングもできることが挙げられます。
まず Java で最初に学ぶ HelloWorld.java を見てみましょう。

```
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("Hello World!!");
  }
}
```

public ？　class ？
static ？　void ？　main ？　String ？　[] ？
System.out.println ？

「Hello World!!」と表示するだけのプログラムですが、そのために呪文のようにクラスの約束事を記述しなければなりませんでした。これが初心者にとってプログラミングを難しいと感じる要因になっています。
Python は右のようにシンプルに記述できます。

```
print("Hello World!!")
```

また Java でプログラムを実行するには、文法的に正しく記述されたソースコードをファイルとして格納し、コンパイルしてバイナリファイルを実行する必要がありました。
Python には REPL 環境（REPL：READ-EVAL-PRINT-LOOP）が用意されていて、１コマンドずつ入力して実行を確認することができます。
昔の BASIC のように手探りで気軽にプログラミングを始められるのです。
他にも文法が簡潔で予約語が少ないことも「習得の容易さ」に寄与しています。具体的には他の言語でよくみられる switch のような多分岐の文が Python にはありません。

「読みやすさ」については、条件判定やループや関数の範囲を示すブロックの見易さが重要です。
多くのプログラミング言語ではブロックを表現するのに波括弧（{}）を使用しますが、プログラムを書く人によって個性が出やすいところです。そのため、コーディング規約等によってインデントを入れることをルール化し「作法」で「読みやすさ」を実現しようとしてきました。
しかし、細かなところでプログラマーのこだわりがあったり、度重なる修正によってズレが生じたり、現実的には徹底できていない状態になることが多くあります。
Python はインデントの深さでブロックを表現するという「文法」によって「読みやすさ」を強制しています。

「効率の良さ」については、Python はデータ処理、通信処理などプログラムでよく使う処理をモジュール化して標準モジュールとして提供しています。これにより非常に少ないコードで効率よく目的とする処理をプログラムすることができます。
また高度な処理を実現するサードパーティ製のパッケージも充実しており、プログラマーのさまざまな要望に対しても十分に応えられます。

そして、MacOS X やほとんどの Linux には標準の状態でインストールされていて、プログラムをすぐに使用できるというメリットもあります。

１．２ Python が注目されている分野

最近書店のプログラミング言語のコーナーで Python 言語の解説や、技術解説で Python をサンプルコードとして掲載している書籍を多く目にします。
書籍やインターネットで Python が注目されている分野のキーワードを列記します。

・ビッグデータ（統計分析、機械学習）
・ＡＩ（ディープラーニング）
・画像処理（OpenCV、自動処理）
・ロボット制御（ROS）
・Web アプリケーション（Django）
・クローリング、スクレイピング
・IoT（RaspberryPi、MicroPytyon など）
・セキュリティツール
・教育（アルゴリズム）
・研究（コンピュータサイエンス、科学技術計算、生物情報学）
・サーバー管理（Fablic）

これだけの分野で注目を集めている背景の一つには、さまざまな分野に特化した大規模なライブラリが充実していることが挙げられます。Pyhton 言語自体は機能を最小限に抑えられているので、足りない部分はライブラリで機能を追加できる仕組みになっています。

教育の分野ではマサチューセッツ工科大学(MIT)やカリフォルニア大学バークレー校などアメリカの大学で言語学習の導入段階において最も採用されています。

企業としては Google 社の主要言語として位置づけられており、Google App Engine でも Python が使われています。また facebook や Dropbox、YouTube、Instagram、BitTorrent なども Python によって開発されています。ここでは RAD（Rapid Application Development）ツールの側面に注目し導入されています。

1．3 Python の種類

Python には動作環境や使用目的によって、いくつかの実装が存在します。

CPython

C 言語で実装されている Python で Python の「標準」という位置付け、一般的に Python といえばこれを示します。
そのため言語仕様や標準ライブラリの最新機能はここの実装から組み入れられます。
公式サイト：https://www.python.org/

Jython

Java で実装された Python で、Java のクラスライブラリが利用でき、Java のガベージコレクションが利用されます。
しかし、最も実行速度が遅い実装です。
公式サイト：http://www.jython.org/

IronPython

C＃で実装されている Python で、.Net Framework および Mono 上で動作し、.NetFramework のクラスライブラリが利用できます。
公式サイト：http://ironpython.net

PyPy

JIT（Just In Time Compiler）による最も実行速度が早い実装です。メモリ使用量を節約できるという特徴もあります。
しかし、x86_32,x86_64,ARM 以外のアーキテクチャでは利用できないという制約があります。
公式サイト：http://pypy.org/

MicroPython

MicroPython は小型マイコンボード用に少ないメモリでも動作するよう最適化された Python 実装です。
もともとは Kickstarter というプロジェクトで pyboard に付属してリリースされたものですが、ESP8266/32 や多くの ARM ベースのアーキテクチャをサポートしています。

1．4 Python のバージョン

現在 Python には 2.7 と 3.7 の２つのバージョンが存在します。
2.x 系は以後メジャーリリースが行なわれる予定はありません。現在開発の主力は 3.x へ移行し積極的に開発が進められています。

2.x から 3.x への主な変更点は、全ての文字列は Unicode になったこと、print 文、exec 文が廃止され関数として扱うようになったこと、切り捨て除算などの言語の幾つかの点が修正されたことです。

１．５ Python の弱点

言語としての Python はシンプルで強力ですが GUI が弱点言えます。
Python には標準で Tkinter という GUI ライブラリが付属していますが、必要最低限の機能しか持たず他の言語の GUI に比べると見劣りします。
GUI は他にもいくつかのサードパーティライブラリによって提供され、次のものがあります。

・PyQt
・wxPython
・PyGTK
・PyGame

どのライブラリも今ひとつ決め手に欠けるのが残念なところです。

他に TUI（Text User Interface）環境の Curses というライブラリもあります。
コマンドラインを多用するサーバーの運用管理用途であれば、有効な選択肢です。

それ以外ではブラウザを GUI 環境として使用するという選択肢もあります。
しかし、Python は Web アプリケーション開発に特化した言語ではありません。
Python には Web アプリケーション開発ライブラリの選択肢が

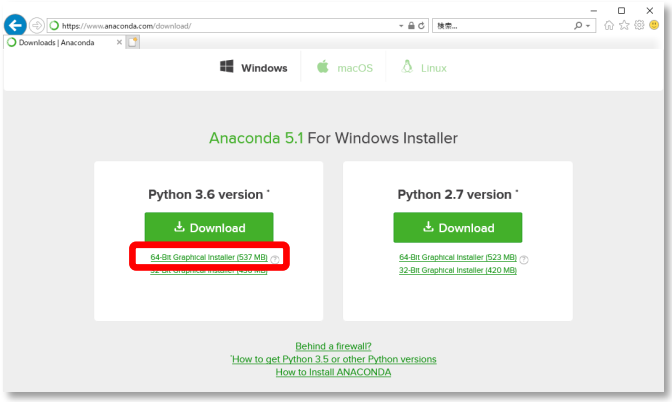
・Turbo Gear
・Django
・Web2py
・Cherrypy
・flask
・bottle

など、多数あります。

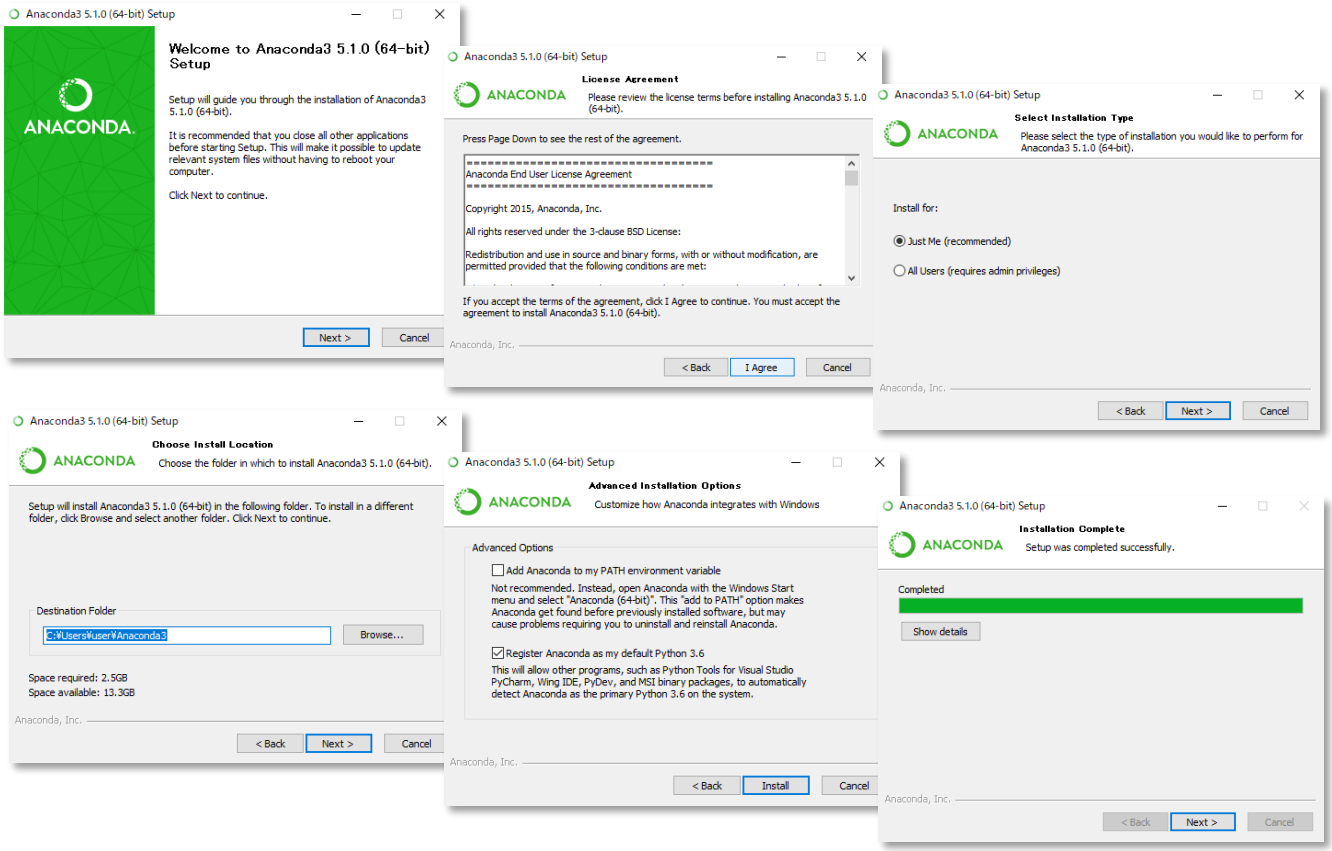
2 章 Python の環境設定

2．1 Python のインストール

本研修では Python3 の学習に Anaconda を使用します。
Anaconda は Python 開発環境やさまざまなライブラリを同梱したディストリビューションです。
https://www.anaconda.com/download/より Windows 版の 64-bit Graphical Installer をダウンロードします。

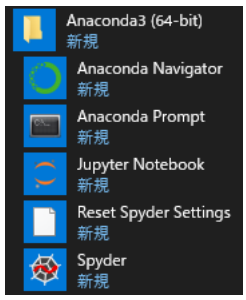


ダウンロードしたファイルを実行します。



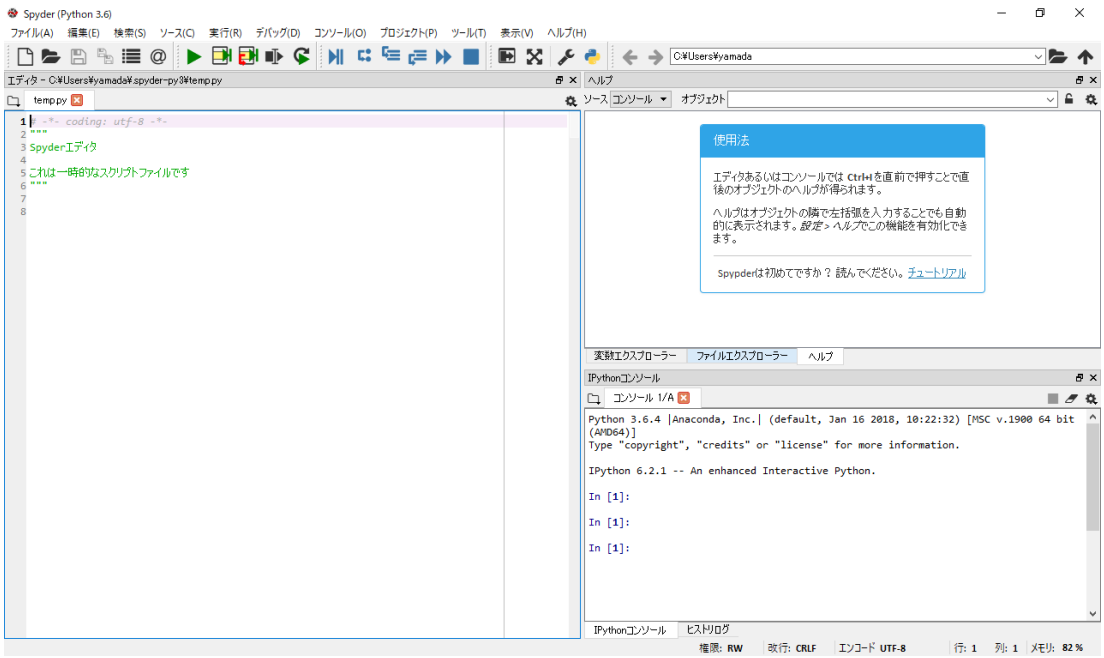
2．2 Python の環境

Anaconda をインストールすると、スタートメニューに次のものが追加されます。



- Anaconda3(64-bit)
  - Anaconda Navigator
    パッケージ管理ツール
  - Anaconda Prompt
    コマンドプロンプト
  - Jupyter Notebook
    ブラウザ上でプログラムを開発する実行する環境
  - Reset Spyder Settings
    Spyder のリセット
  - Spyder
    統合開発環境

Spyder を起動するエディタとコンソールが現れます。



この画面でプログラムのコーディングをします。

Python のスクリプトの先頭には次の行が記述されています。

```
# -*- coding: utf-8 -*-
```

この行はマジックコメントと呼ばれ、このファイルで使用している文字コードを指定します。
この記述が適切でないと実行環境によっては、このコメントやコードの全角文字の部分でエラーが発生することがあります。

それでは試しに次のプログラムを入力してください。

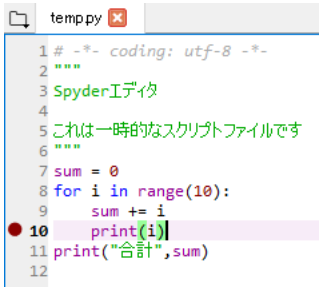
```
sum = 0
for i in range(10):
    sum += i
    print(i)
print("合計",sum)
```

入力ができたら「実行」メニューの「実行」でプログラムを実行します。

```
0
1
2
3
4
5
6
7
8
9
合計 45
```

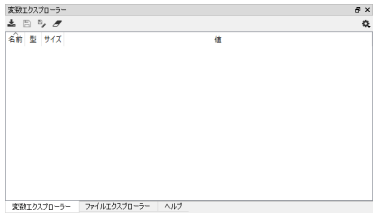
2．3 デバッガの使い方

「2.2 Python の環境」の最後に入力したプログラムファイルを使って解説します。
エディタ上で実行中に停止させたい行を選択し、F12 キーを押下します。



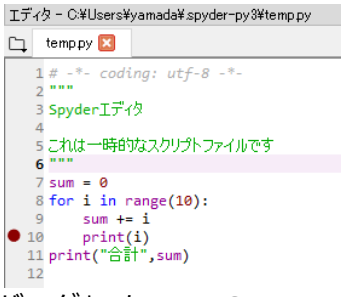
するとブレークポイントが設定され、行番号の左に●のマークが付きます。

次に右側のウィンドウで「変数エスクプローラー」を選択しておきます。



「デバッグ」メニューから「デバッグ」を選択します。
（またはデバッグツールバーからデバッグを実行する）

すると、プログラムの最初のステップで停止します。



停止している行はピンク色で表示されます。

「デバッグ」メニューの Step で 1 行ずつ実行し、変数の状態を「変数エクスプローラー」で確認していきます。

「デバッグ」メニューはそれぞれ次の動作をします。

Step	：現在行を実行し次の行に遷移
Step Into	：関数呼び出しであれば関数の中に遷移
Step Return	：現在の実行している関数を抜けたところまで実行
Continue	：ブレークポイントが設定されている行まで実行
停止	：デバッグを停止

3章 Pythonの文法

3．1 基本文法

ここでは基本的な文法を紹介します。
詳細についてはPythonのホームページを参照してください。（https://docs.python.jp/3/）

コメント

Pythonは「#」以降の行末までをコメントとして扱います。
複数行をまとめてコメントにする場合に、「"""」（ダブルクォーテーションを3つ）もしくは「'''」（シングルクォーテーションを3つ）で複数行の文字列を挟んでコメントとして記述する方法があります。これはドキュメンテーション文字列という書き方で、単なるコメントではなくツールなどから利用される役割を担った文字列です。後述のドキュメンテーション文字列で解説します。

文の終端

一つの文の終わりは改行コードで表現します。
つまり、一行に一つの文を記述するのがPythonの基本です。
しかし、一つの行に文が収まりきらない場合は改行コードの直前に円記号(¥)またはバックスラッシュ(\)を置くことで文を複数に跨いで記述することができます。
また、あまりお奨めしませんが一つの行に複数の文を記述する場合はセミコロン(;)で文の終端を表すこともできます。

変数と値

多くのプログラミング言語と同様にPythonでも変数を利用します。変数名として利用できる文字には以下の決まりがあります。

- 変数名はアルファベット、数字、アンダーバー（_）の組み合わせ
- 始めの1文字は、アルファベットまたはアンダーバー（_）
- アルファベットの大文字と小文字を区別する

変数への値の代入は '=' を使用します。

```
変数名 = 値
```

変数に型の宣言は不要です。代入する値により動的に型を判断します。

変数を削除するにはdel文を使用します。

```
del 変数名
```

定数

Python に定数はありません。
Python プログラマの慣習として、変数名を大文字とアンダーバー(_)のみで構成した場合固定値として扱うというものがあり、定数の代わりに使用しています。

組み込み型

数値

数を表現するために利用するデータ型。
数値の精度によって整数型、浮動小数点型に分かれます。

整数型　　：小数点以下の値を含まない数値
浮動小数点型　：小数点以下の値を含む数値

```
abc = 10
xyz = 3.14
```

文字列型

文字を順番に並べたテキストデータ。（シーケンス型）
"（ダブルクォーテーション）または '（シングルクォーテーション）で囲みます。

```
s = "Hello, World!"
w = "日本語"
```

文字列中のエスケープシーケンスを無効にする raw 文字列型を指定するには「r」を指定します。

```
s = r"price ¥100"
w = r"正規表現で¥d は数字を表します"
```

リスト

複数の要素を順番に並べて格納するためのデータ型。（シーケンス型）
リストには、数値や文字列、リスト自体を含んださまざまな型のオブジェクトを納めることができ、インデックスを使ってリストの要素にアクセスできます。
リストを定義するには、要素をカンマで区切り、その全体を角括弧で囲みます。

```
lst = [ 10,20,30,40,50 ]
```

リストを参照するには変数名に角括弧で囲んだインデックスを指定します。
（インデックスは０からカウントする）

```
lst[2]
```

リストの要素数を調べるには len 関数を使用します。

```
len(lst)
```

リストに要素を追加するには append メソッドを使用する方法、extend メソッドを使用する方法、算術演算子を使用する方法があります。

```
lst.append("apple")
lst += ["pen"]
lst.extend([2,3,4])
lst += [2,3,4]
```

リストから要素を削除するには remove メソッドで要素の値を指定する方法と del 文で要素をインデックス指定する方法があります。

```
lst.remove("apple")
del lst[2]
```

del 文でリスト要素をスライスで削除したり、リスト全体を消去することもできます。

タプル

リストと似ていて複数の要素を順番に並べるためのデータ型。（シーケンス型）
リストと違い要素の追加や変更ができません。
タプルを定義するには、要素をカンマで区切り、その全体を丸括弧で囲みます。

```
tpl = ( 10,20,30,40 )
```

タプルを参照するには、リストと同様に変数名に角括弧で囲んだインデックスを指定します。（インデックスは 0 からカウントする）

```
tpl[2]
```

辞書

リストと同様に、複数の要素を格納することができるデータ型。
要素を順序で管理する代わりに“キー”を使って値にアクセスします。
辞書を定義するには、キーと値をコロンで並べ組とし、その複数の組をカンマで区切り、全体を波括弧で囲みます。

```
dic = { "name":"yamada", "home":"Nagoya", "age":50 }
```

辞書を参照するには、変数名に角括弧でキーを指定します。

```
dic["name"]
```

辞書の要素数を調べるには len 関数を使用します。

```
len(dic)
```

辞書に要素を追加するにはキーを指定して値を代入します。

```
dic["born"] = "Suzuka"
```

辞書から要素を削除するには del 文で要素をキーで指定します。

```
del dic["born"]
```

集合

リストと同様に、複数の要素を格納することができるデータ型。
リストと違い要素の重複を許しません。
集合演算が可能です。
要素に辞書やリストなどの変更可能オブジェクトを指定することができません。
集合を定義するには、要素をカンマで区切り、その全体を波括弧で囲みます。

```
set = { "orange", "apple","pen" }
```

タプル・リスト・集合・辞書まとめ

	タプル	リスト	集合	辞書
括弧	()	[]	{}	{}
データの変更	できない	できる	できる	できる
辞書のキーとして	使える	使えない	使えない	使えない
データの順番	ある	ある	ない	ない
データの重複	できる	できる	できない	キーの重複は できない
要素を空で生成	a = () a = tuple()	a = [] a = list()	a = set()	a = {} a = dict()

演算子

Python では以下の演算子を使用することができます。

算術演算子

a + b	加算
a - b	減算
a * b	乗算
a / b	除算
a // b	切り捨て除算
a % b	剰余
a ** b　　pow(a,b)	a の b 乗

比較演算子

a == b	等しい
a != b	等しくない
a < b	より小さい
a <= b	以下
a > b	より大きい
a >= b	以上
a in シーケンス型	シーケンスに a が含まれる
a not in シーケンス型	シーケンスに a が含まれない
a is b	オブジェクトが同じ
a is not b	オブジェクトが異なる

ブール演算子

a and b	論理積
a or b	論理和
not a	否定

ビット演算子

~a	ビット反転
a & b	ビット論理積
a | b	ビット論理和
a ^ b	ビット排他的論理和
a << b	b ビット左シフト
a >> b	b ビット右シフト

文字列演算子

a + b	文字列連結
a * b	文字列 a を b 回繰り返す

代入演算子

a = b	a に b を代入
a += b	a = a + b と同じ
a -= b	a = a - b と同じ
a *= b	a = a * b と同じ
a /= b	a = a / b と同じ
a %= b	a = a % b と同じ
a **= b	a = a ** b と同じ
a //=b	a = a // b と同じ
a &= b	a = a & b と同じ
a |= b	a = a | b と同じ
a ^= b	a = a ^ b と同じ
a <<= b	a = a << b と同じ
a >>=b	a = a >> b と同じ

制御文

Python では C 言語や Java 言語のように処理のブロックを示す括弧の代わりにインデントを使ってブロックを表現します。

if 文

```
if x < 0:
     x = 0
     print('Negative changed to zero')
elif x == 0:
     print('Zero')
elif x == 1:
     print('Single')
else:
     print('More')
```

ゼロ個以上の elif 部を使うことができ、 else 部を付けることもできます。
elif は「else if」を短くしたもので、過剰なインデントを避けるのに役立ちます。
一連の if ... elif ... elif ... は、他の言語における switch 文や case 文の代用となります。

for 文

Python の for 文は、任意のシーケンス型 (リストまたは文字列) にわたって反復を行います。
反復の順番はシーケンス中に要素が現れる順番となります。

```
a = ['cat', 'window', 'door']
for x in a:
    print(x, len(x))

-----実行結果-----
cat 3
window 6
door 4
```

while 文

while 文は、C 言語と同じく条件が成立している間、後に続くブロックを実行します。

```
cnt = 0
while cnt < 5:
    print(cnt)
    cnt += 1

-----実行結果-----
0
1
2
3
4
```

range() 関数

数列にわたって反復を行う必要がある場合、組み込み関数 range() が便利です。
この関数は算術型の数列を生成します。

```
for i in range(10):
    print(i, end="")
-----実行結果-----
0123456789
```

range(10) は 0 から始まり 10 よりも小さい連続する値を生成します。
range の開始値や、増加量を指定することもできます。

```
for i in range(5, 10):            # 5 6 7 8 9
    print(i)

for i in range(0, 10, 3):         # 0 3 6 9
    print(i)

for i in range(-10, -100, -30):   # -10 -40 -70
    print(i)
```

break 文と continue 文

break 文は、C 言語と同じく、最も内側の for または while ループを中断します。
continue 文は、ループを次の反復処理に実行を移します。

ループの else 節

for や while などのループに「else」文が記述されていた場合は、ループが終了する時に「else」文のブロック内の処理が実行されます。
だだし、break によってループを抜けたときには実行されません。

pass 文

pass 文は何もしません。
pass は、文を書くことが構文上要求されているが、プログラム上何の動作もする必要がない時に使われます。

```
while True:
    pass
```

関数定義

繰り返し使われる処理は関数として定義すると便利です。

def 文

関数を定義する場合は、def 文を使用し引数は括弧内に定義します。

```
def average(a,b,c):
   print((a+b+c)/3.0)

average(11,24,46)
-----実行結果-----
27.0
```

引数の宣言で「引数 = 値」と指定するとデフォルト値を設定できます。

```
def average(a=10,b=20,c=30):
   print((a+b+c)/3)
```

関数の呼び出しには「位置指定引数」と「キーワード引数」の 2 種類の指定方法があります。
キーワード引数とは「引数名＝値」の形で指定する方法です。

```
average(50)                    average(c=15)
-----実行結果-----             -----実行結果-----
33.3333333333                  15.0
```

関数が任意個数の値を受け取れる「可変長引数」という仕組みがあります。
以下のように引数を宣言すると、位置指定引数の呼び出しを「タプル」で受け取ります。

```
def 関数名(*arg):              関数名(10,30,50)
    print(arg)                 -----実行結果-----
                               (10,30,50)
```

また、関数がキーワード引数による任意個数の値を受け取れる「キーワード可変長引数」という仕組みもあります。
以下のように引数を宣言すると、キーワード引数の呼び出しを「辞書」で受け取ります。

```
def 関数名(**arg):             関数名(a=1,b=2,c=3)
  print(arg)                   -----実行結果-----
                               {'a': 1, 'c': 3, 'b': 2}
```

固定長引数と可変長引数を組み合わせもできます。
その場合、固定引数、位置指定可変長引数、キーワード可変長引数の順番で定義します。

```
def 関数名(a,b,*c):
```

```
def 関数名(a,b,**c):
```

```
def 関数名(*a,**b):
```

```
def 関数名(a,*b,**c):
```

return 文

関数の戻り値は return 文によって呼び出し元に返します。

```
def average(a,b,c):
   return (a+b+c)/3
```

```
print(average(11,24,46))
-----実行結果-----
27.0
```

複数の戻り値を返したいときは、タプルで返します。

```
def sumave(a,b,c):
   sum = a+b+c
   return (sum,sum/3.0)
```

```
(total,average) = sumave (11,24,46)
print(total, average)
-----実行結果-----
81  27.0
```

変数のスコープ

変数名が登録される概念的な空間を名前空間といいます。
関数のブロック内部で割り当てた変数名はその関数のブロック内でしか通用しません。そしてその関数の実行が終了すると、その変数は消滅します。
この変数名が登録されている名前空間をローカル名前空間といいます。
この他にソースコードファイル上や対話モードで割り当てた変数名はグローバル名前空間に登録されます。

関数のブロック内部で変数名を参照したとき、以下の順番で変数名の解決を試みます。
（１）実行中の関数内のローカル名前空間
（２）外側の関数のローカル名前空間
（３）グローバル名前空間

関数のブロック内で変数に代入を行うと、その変数名はローカル名前空間に登録されます。
関数のブロック内でグローバル名前空間に登録されている変数名に値を代入すると、ローカル名前空間に同じ名前の変数が登録され、その変数に値が代入されるので要注意です。

グローバル名前空間に登録されている変数に対して値を代入したい場合は、関数の先頭で global 文を使って代入したいグローバル名前空間に登録されている変数を明示的に指定します。

グローバル変数を表示するには globals()関数で、ローカル変数を表示するには locals()関数で可能です。

三項演算子

三項演算子を使うことで代入の条件分岐を一行でスマートに記述することができます

変数＝【条件が True のときの値】 if 【条件】 else 【条件が False のとき値】

例えば、a が 10 以上のときに X に 0 を、そうでないとき X に a-10 を代入する式は以下のように書きます。

```
X = 0 if a >= 10 else a-10
```

書式指定文字列

Python で数値や文字列を様々な書式に変換（フォーマット）するには、３つの方法があります。一つは昔から C 言語の printf 関数などで使用された%書式による書式指定方法、二つ目は Python の文字列クラスの format メソッドによる書式指定方法、そして三つ目は Python3.6 で追加されたフォーマット済み文字列リテラルを使用する方法です。

%書式による書式指定

%書式指定では、「フォーマット文字列」と「差し込みたいオブジェクト」をパーセント（%）で区切って並べます。差し込みたいオブジェクトが複数ある場合は、丸括弧で囲んでタプルで指定する方法と、辞書で指定する方法があります。

フォーマット文字列 % (オブジェクト [,オブジェクト]･･･)
フォーマット文字列 % {キー：オブジェクト [,キー：オブジェクト]･･･}

オブジェクトをタプルで指定したサンプル

```
template = "こんにちは、%s さんの身長は%5.1fcm です。"
data = ("山田",171.5)
print(template % data)
```

オブジェクトを辞書で指定したサンプル

```
template = "こんにちは、%(name)s さんの身長は%(tall)5.1fcm です。"
data = {"name":"山田","tall":171.5}
print(template % data)
```

フォーマット文字列中の「%s」や「%5.1f」、「%(name)s」、「%(tall)5.1f」の部分を「変換指定子」と呼び、対応するオブジェクトの差し込み方法を指定します。

変換指定子の書式は

%[(キー名)][フラグ][桁数][.精度]変換型

キー名とフラグ、桁数、精度は省略可能です。

主な変換型を以下に示します。

変換型	説明
s	文字列（数値なども str()で変換します）
d	符号付き１０進整数
f	１０進浮動小数点数
x	符号付き１６進数（小文字）
X	符号付き１６進数（大文字）
o	符号付き８進数
c	文字一文字

主なフラグを以下に示します。

フラグ	説明
0	数値型に対してゼロによるパディングを行います
-	変換された値を左寄せにします
+	変換の先頭に符号文字 ('+' または '-') を付けます

format メソッドによる書式指定

format メソッドによる書式指定では、「フォーマット文字列」として{}で区切られた置換フィールドを含む文字列を使います。
置換フィールドは位置引数のインデックス、またはキーワード引数の名前を含み、戻り値として format メソッドの引数が置換フィールドに置き換えられた文字列が返されます。

フォーマット文字列.format（オブジェクト [,オブジェクト]･･･）
フォーマット文字列.format（キー＝オブジェクト [,キー＝オブジェクト]･･･ ）

オブジェクトをインデックスで指定したサンプル

```
template = "こんにちは、{0}さんの身長は{1:5.1f}cm です。"
print(template.format("山田",171.5))
```

オブジェクトをキーワード引数で指定したサンプル

```
template = "こんにちは、{name}さんの身長は{tall:5.1f}cm です。"
print(template.format(name="山田",tall=171.5))
```

また、fomant メソッドの引数にタプルや辞書を指定することもできます。

```
template = "こんにちは、{0}さんの身長は{1:5.1f}cm です。"
data = ("山田",171.5)
print(template.format(*data))

template = "こんにちは、{name}さんの身長は{tall:5.1f}cm です。"
data = {"name":"山田","tall":171.5}
print(template.format(**data))
```

置換フィールド中の「：」（コロン）後にあるのは変換指定子で、%書式と類似したものを使用できます。
（詳細は Python ドキュメント » Python 標準ライブラリ » テキスト処理サービス »string --- 一般的な文字列操作 »書式指定ミニ言語仕様を参照のこと）

フォーマット済み文字列リテラル

フォーマット済み文字列リテラルは、接頭辞 'f' または 'F' の付いた文字列リテラルです。これらの文字列には、波括弧{}で区切られた式である置換フィールドを含めることができます。
ただし二重の波括弧 '{{' または '}}' は、それぞれに対応する一重の波括弧に置換されます。

```
name="山田"
tall=171.5
template = f"こんにちは、{name}さんの身長は{tall}cm です。"
print(template)
```

ドキュメンテーション文字列（DocString）

ドキュメンテーション文字列とは、モジュールや関数、クラス、メソッドなどを説明する文字列のことです。モジュールや関数、クラス、メソッドを宣言した後で最初に文字列の形式で説明文を記述するとドキュメンテーション文字列として扱われます。

ドキュメンテーション文字列を付けておくと REPL や Python に対応したエディタなどで、関数の説明を参照したり、ドキュメント生成ツールを利用して自動で関数のマニュアルを生成することができます。
help 関数を使うとドキュメンテーション文字列を参照することができます。

test.py というファイルが以下のような記述だった場合

```
#!/usr/bin/env python
# coding:utf-8

""" このモジュールの説明文です"""

def sample(x,y=100):
   """sample 関数の説明文です"""
   z = x * 100
   print(z)
```

REPL 環境でインポートして help()関数で参照すると

```
>>> import test
>>> help(test)
```

```
Help on module test:

NAME
    test – このモジュールの説明文です

FILE
    /Python3/test.py

FUNCTIONS
    sample(x, y=100)
        sample 関数の説明文です
```

と表示されます。

Python の文字列と日本語

Python2.X の文字列には、通常の文字列であるバイト文字列と Unicode 文字列の２種類があります。（デフォルトではバイト文字列です）
バイト文字列では日本語などのマルチバイト文字の境界を適切に判断できないため日本語の文字列処理で不具合が発生する可能性があります。
Unicode 文字列は１８ビットで１文字を表現するためバイト文字列と比べて表現できる文字数が大幅に増えました。文字列中に別の言語が混在していても、すべて１文字として認識することができるため、日本語の文字列処理も問題なくできます。

バイト文字列を Unicode 文字列に変換

バイト文字列を Unicode 文字列へ変換するには unicode 関数を使う方法とバイト文字列オブジェクトの decode メソッドを使う方法があります。

unicode(バイト文字列 [, エンコード名 [, エラー処理方法]])
バイト文字列.decode([エンコード名 [, エラー処理方法]])

どちらの方法でもバイト文字列をエンコード名で指定された符号化方式とみなして Unicode 文字列に変換します。
エンコード名には'utf-8','shift-jis','shift_jis','sjis','iso-2022-jp','euc-jp'などが使えます。

エラー処理方法

エラー処理方法	説明
strict	例外を発生（デフォルト）
ignore	エラーを無視
replace	エラー箇所を特定文字列で置換

```
str = "本日は晴天なり"
ustr = unicode(str,"utf-8","ignore")
ustr2 = str.decode("utf-8","ignore")
```

Unicode 文字列をバイト文字列へ変換

Unicode 文字列をバイト文字列へ変換するには Unicode 文字列オブジェクトの encode メソッドを使います。

Unicode 文字列.encode([エンコード名 [, エラー処理方法]])

Unicode 文字列をエンコード名で指定された符号化方式のバイト文字列に変換します。

```
ustr = "石の上にも三年"
str = ustr.encode("utf-8","ignore")
```

３．２ 標準関数

Python では特別な宣言をしなくても使用できる標準関数があります。
既に「基本文法」の中で登場した help 関数や range 関数も標準関数です。
ここでは、ここまでの解説で登場していない、けれどもよく使用する標準関数を紹介します。

関数	説明
abs()	数値の絶対値を返します。
bin()	整数を二進文字列に変換します。
chr()	引数に ASCII コードを指定すると該当する文字 1 字からなる文字列を返します。 この関数は ord()の逆です。
float()	引数として数または文字列を受け取ると、浮動小数点数に変換します。
hex()	引数として整数を受け取ると、16 進数文字列に変換します。
input()	この関数は標準入力から行を読み込んで文字列に変換して（末尾の改行を除いて）返します。
int()	引数として数値または文字列を受け取ると、整数に変換します。
len()	指定されたオブジェクトの長さ（要素の数）を返します。
max()	引数にシーケンス型または複数の引数を受け取ると、その中で最大の要素を返します。
min()	引数にシーケンス型または複数の引数を受け取ると、その中で最小の要素を返します。
oct()	整数を８進の文字列に変換します。
ord()	1 文字の Unicode 文字を表す文字列に対し、その文字の Unicode コードポイントを表す整数を返します。
round()	引数を２つ受け取り第一引数を小数点以下第二引数桁で丸めた浮動小数点数の値を返します。
sorted()	引数にシーケンス型を受け取ると要素を並べ替えた新たなリストを返します。
str()	引数のオブジェクトを表示可能な文字列に変換して返します。
sum()	引数にシーケンス型または複数の引数を受け取ると、その総和を返します。
type()	与えられた引数の object の型を返します。

３．３ モジュール

プログラムが長くなるとメンテナンスしやすいように複数のファイルに分割したくなります。
Python はファイルに定義した処理を組み入れて使用する仕組みを持ち、このファイルをモジュールといいます。そしてモジュールはインポートすることで利用できるようになります。

インポート

インポート対象ファイルの探索は次の順序で行われます。

１．実行中のファイルと同じディレクトリ
２．カレントディレクトリ
３．環境変数「PYTHONPATH」に列挙したディレクトリ
４．sys.path に登録してあるディレクトリ

モジュールのインポートは import 文、from～import 文で行います。

import 文

import <モジュール名>でインポートします。<モジュール名>は読み込みたいファイルの拡張子「.py」を除いたものになります。

例えば次の内容が記述されているファイルがカレントディレクトリにあるとします。

```
-----testmodule.py-----
def xxx(n):
    sum = 0
    for i in range(1,n):
        sum += n
    return sum
----------------------------
```

このファイルをモジュールとして組み込んで利用するには次のようにします。

```
import  testmodule
print(testmodule.xxx(5))
```

また、as を使ってモジュール名に別名を定義することもできます。

```
import  testmodule  as  tm
print(tm.xxx(5))
```

from～import 文

from <モジュール名> import <関数名>でインポートします。<関数名>に「*（アスタリスク）」を使用すると、モジュール内の全ての関数をインポートすることになります。
この方式でインポートした場合、使用時にモジュール名の指定は不要になります。

```
from  testmodule  import  xxx
print(xxx(5))
```

引数の取得

Python ではコマンドライン引数は sys モジュールの argv 属性に文字列を要素とするリストとして格納されています。そして、リストの先頭要素(sys.argv[0])はスクリプトファイル名となっています。

dir 関数

dir 関数はモジュール名を引数で指定することで、モジュールの中に定義されている変数、モジュール、関数、その他の、すべての種類の名前のリストを返します。

```
import  sys
dir(sys)
```

引数を指定しない場合は現在のローカルスコープで定義されている名前空間の名前のリストを返します。

```
dir()
['__builtins__', '__doc__', '__name__', '__package__']
```

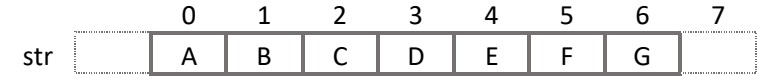
dir()は、組込みの関数や変数の名前は返しません。dir()の引数に__builtins__を指定することで取得することができます。

```
dir(__builtins__)
```

３．４ スライス

スライス表記はシーケンスオブジェクト (文字列、タプルまたはリスト) におけるある範囲の要素を選択します。スライス表記は式として用いたり、代入や del 文の対象として用いたりできます。
例として「ABCDEFG」と言う文字列を使ってスライスの説明を行います。
文字列は右の図のように考えることができます。

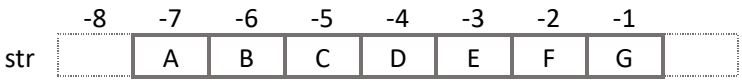
```
str = "ABCDEFG"
```



スライス表記はシーケンスオブジェクトに［開始値：終了値：刻み値］と表現します。（終了値は終了条件で参照されないと考える）
刻み値を省略すると１が指定されたものとします。
刻み値が正であるとき開始値の省略は０、終了値の省略は要素数が指定されたものとします。

```
print(str[:])      # ABCDEFG
print(str[0:7])    # ABCDEFG
print(str[3:])     # DEFG
print(str[:3])     # ABC
print(str[1:6:2])  # BDF
```

また、スライスで指定する値がマイナスの場合は次のように考えます。




```
print(str[-4:])    # DEFG
print(str[-5:-2])  # CDE
print(str[2:-2])   # CDE
```

刻み値が負であるとき開始値の省略は－１、終了値の省略はマイナス（要素数＋１）が指定されたものとします。

```
print(str[::-1])       # GFEDCBA
print(str[-1:-8:-1])   # GFEDCBA
```

３．５ 例外

プログラムで例外を処理するためには try～except 構文を使用します。
try 節のブロックで例外が発生する可能性のある処理を記述し、
except 節には try のブロックで例外が発生した場合に実行される処理を記述します。

基本の書式は以下です

```
try:
  # ここに処理を書く
except:
  # 処理で例外エラーが発生した場合にここの処理をする
```

例えば、数字以外の文字を int()関数で整数化すると以下のように ValueError 例外が発生します。

```
>>> a = int('123b')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '123b'
```

このエラーを捕捉する処理として、例えば数値入力されるまで再入力を促す処理は以下のように書けます。

```
while True:
   try:
     a = raw_input()
     b = int(a)
     break
   except:
     print("再入力してください")
```

また except 節は複数記述することが可能で、except の後に例外エラーを指定することで、特定の例外エラーだけを捕捉することもできます。

```
try:
  # ここに処理を書く
except ValueError:
  # 処理で ValueError 例外エラーが発生した場合にここの処理をする
except:
  # 処理で ValueError 以外の例外エラーが発生した場合にここの処理をする
```

一つの except で複数の例外を捕捉したいときは、例外をタプルの要素として記述することで可能となります。

```
except (例外１,例外２,例外３…):
```

この他にも、処理で例外が発生しなかったときには else 節で処理し、例外が発生するしないに拘わらず実行したいものは finally 節で処理できます。

```
try:
  # ここに処理を書く
except:
  # 処理で例外エラーが発生した場合にここの処理をする
else:
  # 処理で例外エラーが発生しなかった場合にここの処理をする
finally:
  # 処理で例外エラーが発生してもしなくてもここの処理をする
```

例外を意図的に発生させるには raise 文を使います。

```
raise エラー種類（メッセージ）
```

独自の例外エラーは Exception クラスを継承することで作成できます。

```
class MyException(Exception):
  pass

raise MyException("Original Error")
```

３．６ リスト内包表記

リスト内包表記とは、複雑な値を持つリストやタプル・辞書型を手軽に生成するための記法です。

書式

```
[ 式 for 変数 in 対象リスト ]
```

サンプル

```
lst = [x**2+3*x+5 for x in range(10)]
print(lst)
----------------
[5, 9, 15, 23, 33, 45, 59, 75, 93, 113]
```

九九のリストを作成するサンプル

```
[[x*y for x in range(1,10)] for y in range(1,10)]
```

リスト内包表記に条件式を付けることもできます。

書式

```
[ 式 for 変数 in 対象リスト if 条件 ]
```

サンプル

```
lst = [x for x in range(1,40) if x%3 == 0 or x%10 == 3 or x//10 == 3]
print(lst)
----------------
[3, 6, 9, 12, 13, 15, 18, 21, 23, 24, 27, 30, 31, 32, 33, 34, 35, 36, 37,
38, 39]
```