第 2 章 Python 入門

ディープラーニグをはじめとする機械学習を学習し、応用するプログラミング言語としては、Python がよく選ばれます。その理由は、プログラミング言語固有の文法をあまり意識することなく、データやモデルの取り扱いを簡潔に記述できるため、機械学習のアルゴリズムの実装とその検証に集中出来ることや、ディープラーニングを用いたプログラミングを実装するためのライブラリの多くをインポートして利用出来ることがあげられます。

本章では、Python のインストール、Python を実行する環境として Jupyter Notebook のインストール、ディープラーニングのライブラリの一つである TensorFlow 及び共通フレームワークである Keras のインストールを行います。本書では、基本的な流れについてのみ説明します。各プログラムやライブラリは日々改善され、新しいバージョンやライブラリが公開されていますので、詳しくはそれぞれの公式ページを確認しながらインストールし、環境をセットアップすることをお勧めします。

Python	https://www.python.org/
Jupyter Notebook	https://jupyter.org/
TensorFlow	https://www.tensorflow.org/
Keras	https://keras.io/

2.1 Python のインストール

なにはともあれ、Python を使える環境を整えましょう。Python には大きく分けてバージョン 2 系とバージョン 3 系があります。バージョン 2 系と 3 系では、言語仕様と記述方法に違いがあるため、一部のライブラリやサンプルプログラムでは、バージョン 3 系には対応していない場合もありますが、今後はバージョン 3 系に対応したものが増加することが予想されるため、バージョン 3 系を導入します。

Python の環境 <for Win>

本書では、Windows 7 以降(Windows 8, Windows 10)の環境を想定して説明していきます。ちなみに、著者の環境は 2017 年 12 月現在、Windows 7 Enterprise です。

Python の環境は、Windows 環境にはインストールされていません。そのため、まずは Python のセットアップを行います。まずは、Python の公式ページから最新版の Python のセットアップイメージを入手しましょう。

Python	https://www.python.org/

ブラウザで、Python の公式ページを開きます。

Python のインストール

いよいよ、pyenv を利用して、最新の Python 環境を手に入れましょう。まず、最新バージョンがいくつなのか、「pyenv install --list」で確認します。

```
$ pyenv install --list
```

```
$ pyenv install --list
Available versions:
  2.1.3
  2.2.3
  2.3.7
  2.4
  2.4.1
  2.4.2
  2.4.3
  2.4.4
    .
    .
    .
  3.5.4
  3.6.0
  3.6-dev
  3.6.1
  3.6.2
  3.6.3
  3.6.4rc1
  3.7.0a3
  3.7-dev
  anaconda-1.4.0
  anaconda-1.5.0
```

ずーっと下の方を見ると 3.6.3 が最新のようなので、これをインストールします。

```
$ pyenv install 3.6.3
```

インストールが完了したら、このバージョンの Python を利用するように設定しておきましょう。pyenv を利用すると、さまざまなバージョンの Python を切り替えて使えるようになりますが、今回は、この 3.6.3 をデフォルトとして使うようにします。

```
$ pyenv global 3.6.3
```

現在利用中の Python のバージョンを調べたい場合は、以下のようにします。

```
$ python --version
```

```
$ python --version
Python 3.6.3
$
```

2.2 Python の基本文法

この節では、Python の基本文法について説明していきます。本書で説明する Keras を使ったプログラミングで必要になる最低限の文法のみを取り上げます。まずは、コマンドプロンプトで Python の対話モードを起動します。

```
>python
Python 3.6.3 (v3.6.3:2c5fed8, Oct  3 2017, 18:11:49) [MSC v.1900 64 bit
(AMD64)]
 on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Hello, Python!!

では、さっそく「Hello, Python!!」です。print 文を使って次のようにします。

>>> print(“Hello, Python!”)

```
>>> print("Hello, Python!")
Hello, Python!
>>> _
```

変数

次に、Python での変数の宣言方法を学びます。変数というのは、数字や文字を記憶しておく入れ物で、格納されるデータはオブジェクトという概念で構成されています。Python では、他の言語と違って特に変数の型を表す記号は必要なくて、とにかく「変数名 = 値」とすればよいです。変数名に使えるのは、英数字またはアンダーバーで、先頭の文字は英字またはアンダーバーです。

>>> hensu = 1.23456

変数に格納した数値 1.23456 を print 文で表示してみます。

```
>>> hensu = 1.23456
>>> print(hensu)
1.23456
>>> _
```

変数に格納できるデータの型は、数値型(Number)の他に、文字列型(String)などさまざまなデータ型が利用できます。注意したいのは、数値型には、整数(Integer)、浮動小数点(Float)、ブール型(Boolean)などがあることです。

実際にどんなデータ型かは、type 関数で調べることができます。

```
>>> type(hensu)
<class 'float'>
>>> _
```

整数や文字列も調べてみましょう。

```
>>> suuji = 100
>>> type(suuji)
<class 'int'>
>>> hello = “Hello, Python!”
>>> type(hello)
<class ‘str’>
>>> _
```

必要に応じて、型を変換することも可能です。最初に宣言した hensu は浮動小数点型<class ‘float’>でしたが、str 関数を使うと文字列型<class ‘str’>に変換できます。

```
>>> type(hensu)
<class 'float'>
>>> hensu = str(hensu)
>>> type(hensu)
<class 'str'>
>>> _
```

定数

Python では、定数はサポートされておらず、すべて変数として扱われるようです。しかしながら、Python 3 では、None、True、False といった特別な変数だけは、書き換え不可の組み込み定数(Built-in Constants)として用意されています。

```
>>> print(None)
None
>>> print(True)
True
>>> print(False)
False
>>> type(None)
<class 'NoneType'>
>>> type(True)
<class ‘bool’>
>>> type(False)
<class ‘bool’>
>>> _
```

データ型

組み込み型(Built-in Types)として定義されているデータ型には、数値、シーケンス（文字列、リスト、タプルなど）、マッピング（辞書型）、クラス、インスタンス、および例外があります。以下に、よく使う基本的なデータ型をまとめます。

表 2.1.1 Python の主なデータ型

データ型	説明
ブール型	ブール型(bool)は、True または False の 2 値で定義されています。
数値	組み込み型の数値には、整数(int)、浮動小数点(float)などがあります。
文字列	文字列(str)です。(例: ’single’, ”double”, ’’’triple’’’, ”””triple-double”””)
リスト	数字や文字列を順番付きで格納できるリスト(list)です。(例: list = [1, 2, 3])
タプル	書き換え不可なタプル(tuple)、辞書のキーにできる。 (例: tuple = (1, 2, 3))
集合型	重複を許さず順序のないデータ列です。(例: set = {1, 2, 3})

辞書型	Key-Value の辞書(dict)です。(例: dict = {‘one’: 1, ‘two’: 2, ‘three’: 3})

演算

Python では、ほぼすべてのデータ型のオブジェクトについて、比較、判定、文字列への変換が可能です。また、数値型のオブジェクトは、四則演算などの算術演算が可能です。他の言語の経験のある人なら、簡単かもしれません。

```
>>> 1 + 2    # 足し算
3
>>> 3 - 1    # 引き算
2
>>> 4 * 5    # 掛け算
20
>>> 6 / 4    # 割り算
1.5
>>> 6 // 4    # 割り算（整数部）
1
>>> 6 % 4    # 剰余算
2
>>>
```

プログラミング言語によって書式が異なる算術演算や関数の一部を紹介します。

```
>>> abs(-3.14)    # 絶対値を取る
3.14
>>> int(3.14)     # 引き算
3
>>> float(3)      # 浮動小数点への変換
3.0
>>> pow(5, 2)     # 5 の 2 乗
25
>>> 5 ** 2        # 5 の 2 乗
25
>>>
```

リスト型とタプル型

データ型には、リスト型やタプル型があると紹介しました。ここでは、具体的な利用例を紹介します。まずは、リストを宣言して、インデックスを指定した要素の取り出しを行います。リストは、要素を [ ] で囲みます。

>>> list = [要素, 要素, 要素, … , 要素]

```
>>> list = ['アップル', 'ペン', 'ペン', 'パイナップル']
>>> print(list)
['アップル', 'ペン', 'ペン', 'パイナップル']
>>> print(list[1] + list[2])    # 1 番目と 2 番目の要素だけ取り出して連結
ペンペン
>>>
```

一方、タプルは、要素を ( ) で囲みます。

>>> tuple = (要素, 要素, 要素, … , 要素)

```
>>> tuple = ('アップル', 'ペン', 'ペン', 'パイナップル')
>>> print(tuple)
('アップル', 'ペン', 'ペン', 'パイナップル')
>>> print(tuple[1] + tuple[2])  # 1 番目と 2 番目の要素だけ取り出して連結
ペンペン
>>>
```

タプルは追加・削除・変更ができませんが、リストは追加・削除・変更が可能です。

```
>>> list = ['アップル', 'ペン', 'ペン', 'パイナップル']
>>> list.append(100)            # リストの末尾に数値 100 を追加
>>> print(list)
['アップル', 'ペン', 'ペン', 'パイナップル', 100]
>>> del list[2]                 # 2 番目の要素を削除
>>> print(list)
['アップル', 'ペン', 'パイナップル', 100]
>>> list[3] = 'ペン'             # 最後の要素を'ペン'に書換
>>> print(list)
['アップル', 'ペン', 'パイナップル', 'ペン']
>>> list.insert(2, 'ペン')       # 2 番目に'ペン'を追加
>>> print(list)
 ['アップル', 'ペン', 'ペン', 'パイナップル', 'ペン']
>>>
```

この他、リストやタプルでは、番号と要素数を指定することで、データ列の一部をまとめて切り出すスライスと呼ばれる機能や、長さを取得する len 関数などが利用できます。

```
>>> list = ['アップル', 'ペン', 'ペン', 'パイナップル']
>>> print(list[0:2])            # 0 番目から 2 文字取り出す
('アップル', 'ペン')
>>> print(list[1:])             # 1 番目から最後まで取り出す
('ペン', 'ペン', 'パイナップル')
>>> print(len(list))            # list の長さを調べる
4
>>>
```

集合型

リストやタプルによく似たものに集合型(set)があります。集合型は、順番を持たず、重複を許さない配列です。

```
>>> uniq = set(['アップル', 'ペン', 'ペン', 'パイナップル'])
>>> print(uniq)
{'ペン', 'アップル', 'パイナップル'}
>>>
```

集合型への追加・削除・変更の仕方は、リスト型と少し違います。追加には append 関数ではなく add 関数を使い、削除は remove 関数の引数に添字ではなく値を指定します。

```
>>> uniq = set(['アップル', 'ペン', 'ペン', 'パイナップル'])
>>> uniq.add('バナナ')            # uniq に'バナナ'を追加
>>> print(uniq)
{'ペン', 'アップル', 'バナナ', 'パイナップル'}
>>> uniq.remove('ペン')           # uniq から'ペン'を削除
>>> print(uniq)
{'アップル', 'バナナ', 'パイナップル'}
>>> _
```

集合の演算

集合型のデータに任意の値が含まれているかどうかチェックしたいときは、次のようにします。また、２つの集合型データの包含関係、積・和・差分などの演算が可能です。

```
>>> uniq = set(['アップル', 'ペン', 'ペン', 'パイナップル'])
>>> a = set(['アップル', 'ペン'])
>>> b = set(['ペン', 'パイナップル'])
>>> a.issubset(uniq)
>>> True
>>> 'ペン' in a
True
>>> 'バナナ' in a
False
>>> print(a.intersection(b))
{'ペン'}
>>> print(a.union(b))
{'ペン', 'アップル', 'パイナップル'}
>>> print(a.difference(b))
{'アップル'}
>>> _
```

辞書型

リストや集合型を組み合わせても、様々なプログラムが実装可能ですが、Python には、辞書型と呼ばれるデータ型が用意されています。辞書型の要素の１つ１つは、Key と Value で構成され 、Key を指定して Value にアクセスするということが可能になります。

```
>>> dict = {'アップル':1, 'ペン':2, 'パイナップル':1}
>>> print(dict['ペン'])
2
>>> _
```

Key と Value をそれぞれリスト型として取り出すこともできます。

```
>>> dict = {'アップル':1, 'ペン':2, 'パイナップル':1}
>>> print(dict.keys())
dict_keys(['アップル', 'ペン', 'パイナップル'])
>>> print(dict.values())
dict_values([1, 2, 1])
>>> _
```

辞書型への追加・削除・変更の方法は、たとえば次のようなやり方があります。

```
>>> dict = {'アップル':1, 'ペン':2, 'パイナップル':1}
>>> dict['バナナ'] = 5                 # Key を'バナナ'、Value を 5 として追加
>>> print(dict)
{'アップル': 1, 'ペン': 2, 'パイナップル': 1, 'バナナ': 5}
>>> dict.pop('バナナ')                  # Key が'バナナ'の要素を削除
5
>>> print(dict)
{'アップル': 1, 'ペン': 2, 'パイナップル': 1}
>>> dict['ペン'] = 100                  # Key が'ペン'の要素の Value を 100 に変更
>>> print(dict)
{'アップル': 1, 'ペン': 100, 'パイナップル': 1}
>>>
```

文字列の連結と繰り返し

文字列は数値型と違って、プラス「+」で連結されます。

```
>>> apple = 'アップル'
>>> pen = 'ペン'
>>> print(apple + pen)
アップルペン
>>>
```

「join」メソッドでリストを連結することもできます。

```
>>> piko = ['ペン', 'パイナップル', 'アップル', 'ペン']
>>> print(','.join(piko))
ペン,パイナップル,アップル,ペン
>>>
```

また、アスタリスク「*」で繰り返して連結できます。

```
>>> pon = 'ポン'
>>> print(pon * 3)
ポンポンポン
>>>
```

文字列フォーマット

format 関数で、文字列を定式化（フォーマット）できます。文字列や数値で埋めたい部分を「{ }」のように指定しておいて、format 関数の引数に列挙したり、インデックスやキーワードで指定したりもできます。

```
>>> print('{}{}：{}円'.format('アップル', 'ペン', 100))
アップルペン：100 円
>>> print('{1}{2}：{0}円'.format(100, 'アップル', 'ペン'))
アップルペン：100 円
>>> print('{a}{b}：{c}円'.format(a='アップル', b='ペン', c=100))
アップルペン：100 円
>>>
```

シーケンス演算

リスト、タプルのことをシーケンス、集合型のことをセット、辞書型のことをマッピングと呼びます。これらを共通して扱う仕組みをコレクションと呼びます。いくつかの共通した演算が可能です。例えば、len、 max、min などが利用可能です。

```
>>> list = [1, 1, 2, 3, 4, 5]
>>> tuple = (10, 9, 7, 7, 8, 8, 9)
>>> set = {1, -2, 3, -4, 5, -6, 7, -8}
>>> print(len(list), max(list), min(list))     # 長さ、最大、最小を調べて表示
6 5 1
>>> print(len(tuple), max(tuple), min(tuple))  # 長さ、最大、最小を調べて表示
7 10 6
>>> print(len(set), max(set), min(set))        # 長さ、最大、最小を調べて表示
8 7 -8
>>>
```

関数

Python では、関数を定義することができます。何度も同じ処理をしたり、プログラムをわかりやすく整理したりする目的で利用できます。 インデント（字下げ）が必要です。

```
>>> def hello():                          # 関数を定義
...     print('hello, kansu!')            # 関数の中で実行するプログラム
...                                       # 関数の最後で改行
>>> hello()                               # 定義した hello()関数を呼び出し
hello, kansu!
>>>
```

関数には引数を渡すことができます。

```
>>> def hello(who):                      # 引数付き関数を定義
...     print('hello, ' + who + '!')      # 関数の中で実行するプログラム
...                                       # 関数の最後で改行
>>> hello('Python')                       # 引数に'Python'を渡して hello を呼び出し
hello, Python!
>>>
```

引数はさまざまなデータ型で複数個を扱うことができます。また、返値（戻り値）も定義できます。このとき、複数のデータを列挙することでまとめて返すことができます。

```
>>> def hello(who, aisatsu):              # 複数の引数付き関数を定義
...     print(aisatsu + ', ' + who + '!')
...
>>> hello('Python', 'Konichiwa')          # 関数 hello を呼び出し
Konichiwa, Python!
>>> def keisan(x, y):
...     return x, y, x + y
...
>>> e, f, g = keisan(3, 6)
>>> def keisan(x, y):                     # 足し算をする関数 keisan を定義
...     return x, y, x + y                # 引数 x, y と x + y の計算結果を返す
...
>>> e, f, g = keisan(3, 6)                # x, y と x + y の計算結果を e, f, g に入れる
>>> print(e, f, g)                        # e, f, g の中身を print 文で表示
3 6 9
>>>
```

if 文

Python は、他のプログラム言語と同じく if 文が利用できます。もし[条件式]が True だったら[処理 A]を実行する場合の If 文の構文は以下です。Python では、関数の中のプログラムの場合は、if 文のインデントでさらに字下げが必要です。

```
>>> if [条件式] :
>>>     [条件式が True の場合の処理 A]
>>>
```

また、if ~ else のようにして、もし[条件式]が True だったら[処理 A]を実行し、False だったら[処理 B]を実行することもできます。

```
>>> if [条件式] :
>>>     [条件式が True の場合の処理]
>>> else:
>>>     [条件式が False の場合の処理]
>>>
```

このとき、他の言語と少し異なるのは、if 文の条件分岐で実行されるプログラムの行が、必ずインデントされていないといけないという点です。次の例では、if 文を使って数値の大小を比較するプログラムを関数で定義しています。

```
>>> def over(num, thres):
...     if num > thres:               # num を thres と比較する
...         print(str(num) + 'は' + str(thres) + 'より大きい')
...     else:
...         print(str(num) + 'は' + str(thres) + '以下でした')
...
>>> over(333, 100)
333 は 100 より大きい
>>> over(-1, 10)
-1 は 10 以下でした
>>>
```

次のプログラムは、リストに単語が含まれるかどうか、リスト型を集合型に変換してからチェックします。

```
>>> def include(list, keyword):
...     uniq = set(list)
...     if keyword in uniq:           # uniq に keyword が含まれるかチェックする
...         print(keyword + 'が含まれる')
...     else:
...         print(keyword + 'は含まれない')
...
>>> include(['アップル', 'ペン', 'ペン', 'パイナップル'], 'パイナップル')
パイナップルが含まれる
>>> include(['アップル', 'ペン', 'ペン', 'パイナップル'], 'バナナ')
バナナは含まれない
>>>
```

繰り返し文

Python の繰り返し文には、while 文と for 文があります。ある条件に合致する間ずっと繰り返したい場合は while 文、リストなどの要素のそれぞれに対してなど、指定回数だけ処理をしたい場合は for 文を使うとよいようです。

```
>>> while [条件式] :
>>>     [条件式が True の場合に繰り返す処理]
>>>
```

次の例では、num が max より小さい間、変数 num を 0 から 1 ずつ増やします 。

```
>>> def count(max):                    # while 文を使った関数を定義
...     num = 0
...     while num < max:                # num が max より小さい間繰り返す
...         print(num)
...         num += 1                    # num に 1 を足して num に代入
...
>>> count(5)
0
1
2
3
4
>>>
```

for 文を使っても、同様のことが可能です。

```
>>> for [変数] in range([数値]):
>>>     [繰り返す処理]
>>>
```

```
>>> def count(num):                    # for 文を使った関数を定義
...     for i in range(num):            # num の回数繰り返す
...         print(i)                    # 繰り返し回数 i を print 文で表示
...
>>> count(3)
0
1
2
>>>
```

for 文を使うことで、シーケンス、セット、マッピングを順番に取り出して処理することが可能です。機械学習など、データを扱うプログラムでは必要不可欠な機能といえます。

```
>>> for [変数] in [データ列]:
>>>     [繰り返す処理]
>>>
```

```
>>> def every(array):                    # シーケンスの中身を列挙する関数を定義
...     for i in array:                   # array の中身を 1 個ずつ取り出す
...         print(i)
...
>>> every([1, 2, 3])                    # リストを引数に渡す
1
2
3
>>> every((1, 2, 3))                    # タプルを引数に渡す
1
2
3
>>> every({'a':1, 'b':2, 'c':3})        # 辞書型のデータを引数に渡す
a
b
c
>>> every('dog')                        # 文字列を引数に渡す
d
o
g
>>>
```

繰り返しの途中で終了したり、処理をスキップしたりすることもできます。終了する場合は、break 文を使います。break 文の次の行に書かれた処理は実行されず、for 文や while 文の繰り返し処理を抜けてしまいます。

```
>>> for [変数] in [データ列]:
>>>     [実行される処理]
>>>     break
>>>     [実行されない処理]
>>>
```

以下の例では、4 以上の数値の場合に break で抜けます。このため、1, 2, 3 まで表示された後は表示されません。

```
>>> def stop(array):                     # シーケンスの中身を列挙する関数を定義
...     for i in array:                   # array の中身を 1 個ずつ取り出す
...         if i >= 4:                    # i が 4 以上かどうかチェックする
...             break                     # break で for 文を抜ける
...         print(i)
...
>>> stop([1, 2, 3, 4, 5, 1, 2, 3, 4, 5])          # リストを引数として渡す
1
2
3
>>>
```

一方、continue 文を使うと、次の行に書かれた処理をスキップして、繰り返し処理が実行されます。

```
>>> for [変数] in [データ列]:
>>>     [実行される処理]
>>>     continue
>>>     [実行されない処理]
>>>
```

以下の例では、4 以上の数値の場合に continue でスキップします。このため、4 と 5 を表示せず、1, 2, 3, 1, 2, 3 と表示されます。

```
>>> def skip(array):                    # シーケンスの中身を列挙する関数を定義
...     for i in array:                 # array の中身を 1 個ずつ取り出す
...         if i >= 4:                  # i が 4 以上かどうかチェックする
...             break                   # break で for 文を抜ける
...         print(i)
...
>>> skip([1, 2, 3, 4, 5, 1, 2, 3, 4, 5])              # リストを引数として渡す
1
2
3
1
2
3
>>>
```

モジュールの呼び出し

Python の魅力といえばその文法のわかりやすさもありますが、 やはり機械学習やデータ解析、数値計算などに必要なライブラリがたくさん公開されていて、利用可能なことにあります。基本的には、Python でライブラリに含まれるモジュールを利用する際は、以下のように記述してモジュールを import します。

>>> import [モジュール]

例えば、標準ライブラリに含まれる math モジュールを読み込んで、円周率πとネイピア数 e の値を表示してみます。

```
>>> import math
>>> print(math.pi)
3.141592653589793
>>> print(math.e)
2.718281828459045
>>>
```

次に、ディープラーニングや機械学習でよく利用される数式をいくつか試してみます。

```
>>> import math
>>> print(math.pi)
3.141592653589793
>>> print(math.e)
2.718281828459045
>>>
```

また、import したモジュールに別名をつけて利用することもできます。

>>> import [モジュール] as [別名]

```
>>> import math as ma
>>> print(ma.pi)
3.141592653589793
>>> print(ma.e)
2.718281828459045
>>>
```

import したモジュール名をそのまま利用することもできますが、別名をつけるとプログラムを簡潔に書けるようになるため、よく使われます。

以上で Python の基本文法の説明を終了します。いかがですか？少しはわかった気になれたでしょうか？次節以降では、機械学習やディープラーニングを利用するためのライブラリのインストールと使い方の説明をします。