データマイニング入門

Pythonの基礎

式と数学演算子

以下のセルで足し算を計算してみましょう。 `1+1` は式で、単純なプログラム命令となっていて、Python処理系はこの命令を評価して足し算の結果を返しています。セルの式は `Ctrl+Enter` もしくは `Shift+Enter` で実行することができます。

式の中の `1` は値、 `+` は数学演算子です。

In [ ]:
```
# 1+1を計算
1+1
```

数学演算子には他に以下のような演算子があります。これらの演算子を使って式を入力して実行し、その評価結果を確認してみてください。

- `**` ：累乗
- `*` ：掛け算
- `/` ：割り算
- `//` :整数の割り算（小数点以下は切り捨て）
- `%` :割り算の余り
- `+` :足し算
- `-` :引き算

In [ ]:
```
# 2の5乗を計算
2**5
```

通常の数学と同様に、演算子には以下のように優先順序（評価の順序）があります。括弧 `()` を使うことで、明示的に評価の順序を変えることもできます。 `** > *, /, //, % > +, -`

In [ ]:
```
(1+1)**(2+3)
```

以下の式は間違った命令で `Syntax Error` （文法エラー）となります。エラーの対処については後で学びます。

In [ ]:
```
1+*
```

型

値（例えば先の式の `1` ）のタイプをデータ型と呼びます。すべての値はいずれかのデータ型に属しています。以下にPythonの主なデータ型を示します。

- 整数(int): -1, 0, 1
- 浮動小数点(float): -1.0, 0.0, 3.14,
- 文字列（str）: 'a', '1', 'U Tokyo'

浮動小数点型は小数点を含む数値です。文字列型は文字・テキストの値です。文字列は、シングルクオート（ `'` ）もしくはダブルクオート（ `"` ）で囲んで記述します。 `''` や `""` のように空の文字列も値となります。

演算子の意味は、一緒に使われる値のデータ型によって変化します。整数型や浮動小数点の値に対しては、 `+` は値の足し合わせを意味します。一方、文字列の値に対しては、 `+` は文字列をつなぐ文字列連結演算子となります。

In [ ]:
```
'Hello' + ' ' + 'World!'
```

`*` は整数型や浮動小数点の値の掛け算を意味しますが、文字列と整数と併せて用いると文字列の複製演算子となります。以下の式を評価すると、整数で指定した回数だけ文字列を繰り返した文字列になります。

In [ ]:
```
# Helloを5回繰り返し出力
'Hello'*5
```

以下の式はそれぞれ評価されるでしょうか？

In [ ]:
```
'Hello'+1
```

In [ ]:
```
'Hello'*'World!'
```

In [ ]:
```
'Hello'*1.0
```

変数

変数は値や式の評価結果を格納するための"箱"の役割を果たします。値や式の評価結果を変数へ保存することで、その結果を再利用することができます。変数に値を保存するには、以下のような代入文を用います。

```
変数名 = 値 or 式
```

In [ ]:
```
# 変数xを初期化
x=1

# 変数yを初期化
y=1

# xとy利用して変数zを初期化
z = x+y
z
```

In [ ]:
```
# zを上書き
z = z+1
z
```

In [ ]:
```
# 変数word1を初期化
word1='Hello'

# 変数word2を初期化
word2='World'

# word1とword2を利用して変数sentenceを初期化
sentence = word1 + ' ' + word2
sentence
```

In [ ]:
```
#sentenceを上書き
sentence = sentence+'!'
sentence
```

コーディングではわかりやすい変数名をつけましょう。

変数の命名規則

- 空白を含まない
- 文字と数字と下線記号_から構成される
- 数字から始まらない
- 特殊文字（$や`など) を含まない

print 関数

print() 関数を使うことで、以下のように括弧内の文字列を出力させることできます。この時、Pythonは print() 関数を呼び出し、'Hello World!'という文字列の値を関数に渡しています。関数については後で学びます。なお、シングルクオートは文字列の開始と終了を表す記号で、文字列の値には含まれないので表示されません。

In [ ]:
```
print('Hello World')

# 変数greetに文字列を代入
greet='Hello World'
print(greet)
```

整数型や浮動小数点型の値を文字列型の値とともに print() 関数に渡したいときは、 str() 関数を使って文字列型に変換します。

In [ ]:
```
print('int:'+str(1) + " float:"+str(3.14))
```

ブール型と比較演算子

ブール型は `True` または `False` の2種類の値をとります。以下に示す比較演算子は、2つの値を比較して1つのブール型の値を返します。

- `==` ： 等しい
- `!=` ：等しくない
- `>` ：より大きい
- `<` ：より小さい
- `>=` ：以上
- `<=` ：以下

`>,<,>=,<=` の比較演算子は、両辺が同じ型どうしである必要があります。

In [ ]:
```
1 == 1
```

In [ ]:
```
1 == 0
```

In [ ]:
```
1 == 1.0
```

In [ ]:
```
'Hello' == 'Hello'
```

In [ ]:
```
'Hello' == 'hello'
```

In [ ]:
```
1 == '1'
```

In [ ]:
```
x = 1
x > 0
```

In [ ]:
```
x = -1
x > 0
```

ブール演算子

ブール演算子は、ブール型の値を組み合わせる場合に使います。比較演算子のように1つのブール型の値を返します。

- and
- or
- not

二項ブール演算子

and と or 演算子は、常に2つのブール値（もしくは式）をとるので、二項演算子と呼ばれます。 and 演算子は、以下のように、2つのブール値が True の時のみに True となり、それ以外は False となります。

- True and True：True
- True and False：False
- False and False：False
- False and False：False

一方、 or 演算子は、以下のように、2つのブール値がどちらかが True なら True となり、両方が False なら False となります。

- True or True：True
- True or False：True
- False or True：True
- False or False：False

not 演算子

not 演算子は1つのブール値（もしくは式）をとり、そのブール値を以下のように反転させます。

- not True：False
- not False：True

ブール演算子と比較演算子

比較演算子はブール値を返すので、以下のようにブール演算子と組み合わせて使うことができます。

In [ ]:
```
(0<1) and (1<2)
```

In [ ]:
```
x = 1
y = -1
(x>0) and (y>0)
```

In [ ]:
```
x=1
y=-1
(x ==1) or (y==1)
```

複数の演算子がある時は、算術演算子と比較演算子を評価した後に、まず not 演算子を評価し、次に and 演算子、最後に or 演算子を評価します。

In [ ]:
```
x = 1
y = -1
x+y==0 and not x-y< 0 and (x >0 or y > 0)
```

フロー制御

条件式

ブール演算子を使った式は、条件式ともよばれます。条件式は、これから説明するフロー制御文（ `if` 文など）で使われます。条件式は、常に1つのブール値に評価され、その値が `True` か `False` によって、フロー制御文は次に何を実行するかを決定します。

コードブロック

Pythonのコードは1行以上をひとまとまりとしてブロックとすることができます。ブロックの区間は、コードのインデントで指定します。インデントとは、行の先頭に何個かのスペースを入れることです。Pythonの標準コーディングスタイルでは、4文字のスペースを入れます（ノートブックのセルではtabでインデントが挿入されます）。

- ブロックはインデントで始まる
- ブロックの中には新しいブロックを含めることができる
- インデントがなくなるか、上位のブロックのインデントに戻るとそのブロックは終了する

```
x=1
if x >= 0:
    if x == 0:  # 第1ブロック開始
        print('0')  # 第2ブロック開始
    else:  # 第2ブロック終了
        print('positive')  #第3ブロック開始
```

`if` 文

`if` 文は最もよく用いるフロー制御文の1つです。 `if` 文に続くブロックは、 `if` 文の条件式が `True` の時に実行されます。条件式が `False` ならば、そのブロックの実行はスキップされます。フロー制御文はすべてコロン（ `:`)で終わり、次にコードのブロックが続きます。

`else` 文

`if` 文には、オプションとして `else` 文を続けることができます。 `else` 文に続くブロックには、先の `if` 文の条件式が `False` の時に実行されます。 `else` 文には条件式は必要ありません。

---

elif 文

複数のブロックからいずれか1つを実行したい場合は、 if 文に elif 文を続けて、それ以前の条件式が False だった場合に、別の条件式を判定させます。この時、 elif 文に続くブロックには、 elif 文の条件式が True の時に実行されます。

```
if 条件式1:
    ブロック1
elif 条件式2:
    ブロック2
elif 条件式3:
    ブロック3
else:
    ブロック4
```

In [ ]:
```
# 変数ageを初期化
age = -1

# ageの値で条件分岐
if age == 20:
    print('come of age')
elif age > 65:
    print('elder')
elif age > 20:
    print('adult')
elif age >= 0:
    print('child')
else:
    print('before birth')
```

while 文

while 文を使うと、 while 文の条件式が True である限り、 while 文に続くブロックを何回も繰り返すことができます。 while 文のブロックの終わりでは、プログラムの実行は while 文の最初に戻り、繰り返し while 文の条件式を判定します。条件式が False になると、ブロックの実行をスキップして繰り返しを抜けます。

```
while 条件式:
    ブロック
```

In [ ]:
```
# 変数countを初期化
count=1

# countが5以内であればwhile文内の処理を繰り返す
while count <=5:
    print(count)
    count = count+1
```

break 文

`break` 文を使って、 `while` 文の繰り返しブロックから抜け出すことができます。Jupyter Notebookで無限ループに入りセル操作が受け付けられない時は、メニューのKernelからJupyter NotebookをRestartしてください。

In [ ]:
```
# 変数countを初期化
count=1
while True:
    # countが5を超えたらwhile文の処理を抜ける
    if count > 5:
        break
    print(count)
    count = count+1
```

for文

`while` 文では条件式が `True` のあいだ、ブロックの実行が繰り返されます。一定の回数だけ、ブロックの実行を繰り返したい場合は、 `for` 文と `range()` 関数を用います。

```
for 変数名 in range()関数:
    ブロック
```

`range()` 関数では、第1引数で繰り返し変数の開始値を指定し、第2引数では終了値より1大きい数を指定します。 `for` に続く変数には、繰り返しごとに `range()` 関数が返す値が代入されます。

In [ ]:
```
# 変数countに1から5を順に代入してfor文内を繰り返す
for count in range(1,6):
    print(count)
```

関数

関数はプログラムの処理や手続きの流れをまとめて小さなプログラムのようなものです。関数の主な目的は、頻繁に呼び出されるような処理のコードをまとめることです。

`print()` や `range()` はPythonが持つ組み込み関数ですが、自分で関数を定義することもできます。以下は、'Hello'を出力するだけの単純な関数を定義しています。1行目では `def` で `hello()` という名前の関数を定義することを宣言しています。 `def` 文に続くブロックが関数の本体です。このブロックは関数が呼び出された時に実行されます。関数を定義すると、その関数名を使った関数を呼び出すことができます。

In [ ]:
```
def hello():
    # Helloを出力
    print('Hello')

hello()
```

引数

関数を定義する際に、括弧の中に関数へ渡す変数の一覧を記述することができます。これらの変数は関数のローカル変数となります。ローカル変数とはプログラムの一部（ここでは関数内）でのみ利用可能な変数で、関数の外からは参照することはできません。また、関数が呼び出された後はその変数は消滅します。

In [ ]:
```
def hello(greet):
    # 引数greetの値を出力
    print(greet)

hello("Hello")
```

戻り値

関数は受け取った引数を元に処理を行い、その結果の戻り値を返すことができます。戻り値は、 `return` で定義します。関数の戻り値がない場合は、 `None` が返されます。

In [ ]:
```
def hello(greet):
    # 引数greetの値を返す
    return greet

echo = hello("Hello")
print(echo)
```

複数の引数

関数は任意の数の引数を受け取ることができます。複数の引数を受け取る場合は、引数をカンマで区切ります。これらの引数名は重複しないようにしましょう。

In [ ]:
```
def hello(en, fr, de):
    print(en, fr, de)

hello("Hello","Bonjour","Guten Tag")
```

キーワード引数

一般的な引数（位置引数とも呼ばれます）では、事前に定義した引数の順番に従って、関数は引数を受け取る必要があります。キーワード引数を使うと、関数は引数の変数名とその値の組みを受け取ることができます。その際、引数は順不同で関数に渡すことができます。

In [ ]:
```
def hello(en, fr, de):
    print(en, fr, de, sep=', ')

hello("Hello", de="Guten Tag", fr="Bonjour")
```

リスト

リストは値を格納するためのデータ構造の1つです。リストまたはタプルは複数の値を格納でき、大量のデータを保持しながら処理するプログラムを書くのに役立ちます。また、リストの中に他のリストを入れ子に含むこともできるので、階層的なデータ構造を表現するのにも使えます。

リストは以下のように、複数の値（要素と呼びます）をカンマ区切りにして、四角括弧[]で囲って記述します。リストはそれ自体が値なので、他の値のように、変数に代入したり、関数に渡すことができます。なお、関数にリストを渡して操作すると、その操作は元のリストに反映されることに注意してください。

`[]` という値は空リストと呼び、要素が1つも入っていないリストを表します。

In [ ]:
```
num=[1,2,3,4,5]
num
```

In [ ]:
```
greet=['hello','bonjour','guten tag']
greet
```

In [ ]:
```
def hello(greet_list):
    print(greet_list)

hello(greet)
```

リストとインデックス

以下では、 `ut_dep` という変数に `['理1','理2','理3','文1','文2','文3']` という文字列の要素からなるリストが入っています。ここで、 `ut_dep[0]` を評価して値を見てみましょう。同様に、 `ut_dep[1]` を評価するとどうなるでしょう。

In [ ]:
```
ut_dep=['理1','理2','理3','文1','文2','文3']

# リストの先頭要素を出力
print(ut_dep[0])

# リストの2番目の要素を出力
print(ut_dep[1])
```

リストに続く四角括弧内の整数はインデックスと呼ばれ、リストの各要素に対応しています。リストの先頭の要素のインデックスは0、2番目の要素のインデックスは1、3番目は2となります。このようにインデックスを使うことで、リスト内の任意の要素にアクセスできます。

In [ ]:
```
print('文科:'+ut_dep[3]+","+ut_dep[4]+","+ut_dep[5])
```

インデックスは0から始まりますが、インデックスとして負の整数も用いることができます。インデックス-1はリストの末尾の要素に対応し、-2は末尾から2番目の要素に対応しています。

In [ ]:
```
print('文科:'+ut_dep[-3]+","+ut_dep[-2]+","+ut_dep[-1])
```

リストのは他のリストを入れ子に入れることができます。リストの中のリストの要素にアクセするには以下のように複数のインデックスを使います。第1のインデックスにはどのリストを用いるのかを指定し、第2のインデックスにはそのリストの中の要素に対応するものを指定します。

In [ ]:
```
ut_dep2=[['理1','理2','理3'],['文1','文2','文3']]
print('3類:'+ut_dep2[0][2]+","+ut_dep2[1][2])
```

リストのインデックスを用いるとリストの要素の値を変更できます。リストの要素を削除する場合は `del` 文を使います。削除した要素より後ろの要素は、ひとつずつ前にずれます。

In [ ]:
```
ut_dep=['理','文']
ut_dep[0]='Science'
ut_dep[1]='Human'
print(ut_dep)

del ut_dep[0]
print(ut_dep)
```

リストとスライス

インデックスではリストの任意の要素にアクセスできましたが、スライスを用いるとリスト内の複数の要素にアクセスすることができます。スライスでは四角括弧の中に、2つの整数をコロンで区切って記述します。第1番目の整数はスライスの開始インデックスを、第2整数はスライスの終端インデックスを表しますが、そのインデックス自身は含まれずに、それより1つ小さいインデックスまでを含みます。スライスを評価すると新しいリストとなります。

In [ ]:
```
ut_dep=['理1','理2','理3','文1','文2','文3']

# リストの先頭から3番目までの要素
ut_science = ut_dep[0:3]

#リストの4番目から6番目までの要素
ut_human = ut_dep[3:6]

print(ut_science)
print(ut_human)
```

スライスで第1のインデックスを省略するとインデックス0を指定したのと同じになり、リストの先頭からスライスとなります。また、第2のインデックスを省略するとリストの長さを指定したのと同じになり、リストの末尾までのスライスとなります。なお、リストの長さは `len()` 関数で取得できます。

In [ ]:
```
ut_dep=['理1','理2','理3','文1','文2','文3']
print(len(ut_dep))

# リストの先頭から3番目までの要素
ut_science = ut_dep[:3]

#リストの4番目から終端までの要素
ut_human = ut_dep[3:]

print(ut_science)
print(ut_human)
```

リストの連結

2つのリストに + 演算子を適用すると、2つのリストが連結され新たに1つのリストが生成されます。また、 * 演算子をリストと整数に適用すると、リストを整数の数分だけ複製します。

In [ ]:
```
ut_science=['理1','理2','理3']
ut_human=['文1','文2','文3']
ut_dep = ut_science+ut_human
print(ut_dep)
print(ut_dep*2)
```

リストと for 文

さきほど出てきた `for` 文の繰り返しは、リストのように複数の要素を持つデータ構造から要素を1つずつ取り出してコードブロックを繰り返しています。そのため、リストと `for` 文を組み合わせると、リストの要素を1つずつ取り出しながら処理することができます。

In [ ]:
```
# リストの要素を1つずつ取り出し
ut_dep=['理1','理2','理3','文1','文2','文3']
for dep in ut_dep:
    print(dep)
```

リストのインデックスを用いて、繰り返し処理をしたい場合は、 `range()` 関数と `len()` 関数を用いて以下のように記述します。

In [ ]:
```
ut_dep=['理1','理2','理3','文1','文2','文3']
for i in range(len(ut_dep)):
    print(i,ut_dep[i])
```

リストと in 演算子

in 演算子を使うと、ある要素がリストの中に含まれているかどうかを判定することができます。 in は式として用い、調べたい要素と対象のリストの間に書きます。この式を評価するとブール値となります。

In [ ]:
```
ut_dep=['理1','理2','理3','文1','文2','文3']
'理1' in ut_dep
```

リストとメソッド

メソッドは、あるオブジェクト（リストはオブジェクトの1つです）について呼び出し可能な専用の関数です。リストのようなデータ型には1連のメソッドが備わっていて、例えばリスト型には、検索（index()）、追加（append(), insert()）、削除（remove()）、並び替え（sort()）、取り出し（pop()）などのリストの要素を操作するための便利なメソッドが準備されています。

index()メソッド

index() メソッドは、以下のように値を渡すと、リストの要素からその値のインデックスを返します。リストの中に値がなければ、 ValueError を返します。リストの中に値が複数存在している場合は、最初に出現した方のインデックスを返します。

In [ ]:
```
ut_dep=['理1','理2','理3','文1','文2','文3']
ut_dep.index('理2')
```

apped()メソッドとinsert()メソッド

append() メソッドや insert() メソッドを使うと、以下のようにリストに新しい要素を追加することができます。 append() メソッドは、引数に渡された値をリストの末尾に要素として追加します。 index() メソッドでは、第1引数で値を挿入するインデックスを指定することで、リストの任意の場所に要素を追加することができます。

append() メソッドや insert() メソッドは元のリストを書き換える操作をしていて、戻り値で新しいリストが返るわけではないことに注意してください。

In [ ]:
```
ut_dep=['理2','理3','文1','文2']

# リストの末尾に要素を追加
ut_dep.append('文3')
print(ut_dep)

# リストの先頭に要素を追加
ut_dep.insert(0,'理1')
print(ut_dep)
```

remove()メソッド

remove() メソッドは、渡された値をリストから削除します。リストに含まれない値を削除しようとすると、 ValueError を返します。リストの中に値が複数含まれる場合は、最初のあたいだけが削除されます。

del 文はリストから削除したい値のインデックスがわかっている時に使い、 remove() メソッドはリストから削除したい値がわかっている時に使います。

In [ ]:
```
ut_dep=['理1','理2','理3','文1','文2','文3']
ut_dep.remove('文1')
print(ut_dep)
```

pop()メソッド

pop() メソッドは、渡された値のインデックスの要素をリストから削除し、削除した値を返します。 pop() メソッドは、リストから任意の要素を取り出したい時にも使えます。

In [ ]:
```
ut_dep=['理1','理2','理3','文1','文2','文3']
print(ut_dep.pop(3))
print(ut_dep)
```

sort()メソッド

sort() メソッドを使うと、リストの要素（数や文字列）を並べ換えることができます。

append() メソッドや insert() メソッドと同じく sort() メソッドは元のリストを書き換える操作をしていて、戻り値で新しいリストが返るわけではないことに注意してください。

In [ ]:
```
num = [4,3,5,1,2]
num.sort()
num
```

In [ ]:
```
word = ['desert','banana','carrot','apple']
word.sort()
word
```

文字列の並び替えは、ASCIIコード順で行われます。アルファベット順で行いたい場合は、キーワード引数keyにstr.lowerを指定します。

In [ ]:
```
word = ['Desert','Banana','carrot','apple']
word.sort()
print(word)
word.sort(key=str.lower)
print(word)
```

辞書

辞書は、リスト同様に複数の要素の集合を格納するためのデータ構造の1つです。リストのインデックスが整数型だったのに対して、辞書のインデックスにはさまざまなデータ型（整数、浮動小数点数、文字列など）を用いることができます。辞書のインデックスをキーと呼び、キーには対応する値が存在します。リストと同様に辞書のキーとして整数値を使うこともできます。その場合は、任意の整数値をとることができます。

辞書は以下のように、複数の要素（キーと値の組み）をカンマ区切りにして、波括弧{}で囲って記述します。ここでは、'semester', 'title', 'year', 'unit'というキーが、それぞれ'S', 'data mining', '2018', 2という値と対応している辞書を変数 `course` に代入しています。

In [ ]:
```
course = {'semster':'S', 'title':'data mining', 'year':'2018',  'unit'
```

辞書のキーを使うことで、以下のようにキーに対応する値にアクセスすることができます。

In [ ]:
```
print(course['title'] + ', ' + course['year']+course['semster'])
```

In [ ]:
```
# キーと値を更新
course['year']=2017
course['semster']='A'
del course['unit']
course
```

辞書とkeys, values, itemsメソッド

リストと異なり、辞書の要素には順序がありません。リストの先頭はインデックスが0で、末尾はインデックスが-1でしたが、辞書は先頭あるいは末尾の要素という概念がありません。また、2つの辞書の中身が同じか判定する際も、辞書では要素の順番は影響しません。このように辞書には要素の順序関係がないので、インデックスやスライスで部分を抜き出すことはできないことに注意してください。

辞書にはインデックスやスライスで要素にアクセスできませんが、キー、キーに対応する値、キーと値の組み、それぞれにアクセスするためのメソッドがあります。

- keys()：辞書のすべてのキーをリストで取得
- values()：辞書のすべての値をリストで取得
- items()：辞書のすべてキーと値の組みをリストで取得

これらのメソッドは以下のようにfor文と一緒に使うことができます。

In [ ]:
```
urse = {'semster':'S', 'title':'data mining', 'year':'2018',  'unit':2}
r k in course.keys():
  print(k)

r v in course.values():
  print(v)

r i in course.items():
  print(i)

r k, v in course.items():
  print(k, v)
```

辞書と in 演算子

リストにある要素が含まれるかどうか判定するのに、 in 演算子を使ったように、辞書にあるキーや値が存在するかを判定するのにも、 in 演算子を使うことができます。

In [ ]:
```
urse = {'semster':'S', 'title':'data mining', 'year':'2018',  'unit':2}

キーに'year'があるか?
int('year' in course.keys())

値に'year'があるか?
int('2017' in course.values())
```

辞書のメソッド

リスト型でみたメソッドのように、辞書型にも辞書を操作するための便利なメソッドが準備されています。さきほどの `keys`, `values`, `items` メソッドはそのようなメソッドです。

get()メソッド

`get()` メソッドを使うと、辞書にキーの存在の確認ができます。第1引数には存在を確認したいキー、第2引数にはそのキーが存在しない時に用いる値を渡します。

In [ ]:
```
course = {'semster':'S', 'title':'data mining', 'year':'2018'}
print(course.get('year',2017))
print(course.get('unit',1))
```

pop()メソッド

リストと同じく、 `pop()` メソッドを使って、渡されたキーに対応する要素を辞書から削除し、削除した要素の値を返します。 `pop()` メソッドは、辞書から任意の要素を取り出したい時にも使えます。

In [ ]:
```
course = {'semster':'S', 'title':'data mining', 'year':'2018',  'unit'
print(course.pop('unit'))
print(course)
```

文字列

リストと文字列

文字列は1文字の要素が並んだリスト（正確にはタプル）とみなすことができます。そのため、インデックスによる要素の指定、スライスによる部分の取り出し、 `for` 文での要素の繰り返し処理、 `len()` 関数による長さの取得、 `in` 演算子による要素の検索など、リストに対して可能なことの多くは文字列に対しても可能です。

リストは変更可能なデータ型ですが、文字列はそれ自体が変更不可能です。文字列のように要素を変更したり追加したり削除したりできなくしたリストをタプルと呼びます。

リスト型や辞書型と同様に、文字列型にも文字列を操作するための便利なメソッドが準備されています。

In [ ]:
```
course = 'datamining'

print(course[0])
print(course[-1])
print(course[:4])
print(course[4:])

for i in course:
    print(i)

for i in range(len(course)):
    print(i,course[i])

print('data' in course)
```

upper(), lower()メソッド

`upper()`, `lower()` メソッドは、元の文字列のすべての文字を、それぞれ大文字または小文字に変換した文字列を返します。なお、以下のように `ut.upper()` や `ut.lower()` だけでは、元の文字列 `ut` は変更されません。 `ut = ut.lower()` として変数に代入することで、 `ut` を変更します。

In [ ]:
```
ut = 'The University of Tokyo'
print(ut.upper())
print(ut)
ut = ut.lower()
print(ut)
```

join(), split()メソッド

join() メソッドは、文字列のリストを渡すと、リスト中の文字列を元の文字列で連結したものを返します。以下では、リスト中の文字列を' : 'で連結しています。

In [ ]:
```
' : '.join(['理1','理2','理3','文1','文2','文3'])
```

split() メソッドは、文字列を渡すと、その文字列で、元の文字列を分割してリストとして返します。以下では、文字列を' : 'で分割してリストにしています。

In [ ]:
```
'理1 : 理2 : 理3 : 文1 : 文2 : 文3'.split(' : ')
```

count()メソッド

count() メソッドは、文字列を検索して、引数で指定した文字列が含まれている個数を返します。

In [ ]:
```
'the university of tokyo'.count('t')
```

replaceメソッド

replace() メソッドは、引数で指定した文字列で、元の文字列の指定部分を置き換えた新しい文字列を返します。元の文字列を直接置き換えないことに注意してください。

In [ ]:
```
s = 'I love Tokyo'
print(s.replace('Tokyo', 'Paris'))
print(s)
```

ファイルの入出力

Pythonでファイルを読み書きするには以下のステップがあります。なお、ここではテキストが含まれるプレーンテキストファイルの読み書きを扱うこととします。

1. `open()` 関数を呼び出し、 `File` オブジェクトを取得する
2. `File` オブジェクトの `read()` や `write()` メソッドを呼び出して読み書きする
3. `File` オブジェクトの `close()` オブジェクトを呼び出してファイルを閉じる

open() 関数

`open()` 関数を使ってファイルを開くには、開きたいファイルへのパスを文字列として渡します。パスは絶対パスでも相対パスでも構いません。パスは例えば、以下のように指定します。

```
# 同じフォルダ（ディレクトリ）ファイルがある場合
data_file = open('data.txt')

# 同じフォルダの下位フォルダ、フォルダ名：data、にファイルがある場合
data_file = open('data/data.txt')

# 上位のフォルダ、フォルダ名：data、にファイルがある場合
data_file = open('../data/data.txt')
```

`open()` 関数を呼び出すと、ファイルを読み込みモードで開くことになります。この時、ファイルからは読み込みだけが可能になり、書き込んだり変更したりすることはできません。

`open()` 関数の戻り値は `File` オブジェクトです。 `File` オブジェクトはデータ型の一つで、ファイルを操作するためのメソッドが用意されています。

read()メソッド, readlines()メソッド

`File` オブジェクトの `read()` メソッドを使うと、ファイル全体をひとつの文字列として読み込むことができます。 `read` メソッドは、ファイルの内容をひとつの文字列として返します。

In [ ]:
```
infile = open('data_mining.txt')
content = infile.read()
print(content)
```

File オブジェクトの readlines() メソッドを使うと、1行ずつの文字列を要素とするリストとしてファイルを読み込むことができます。

In [ ]:
```
dmfile = open('data_mining.txt')
content = dmfile.readlines()
print(content)
# '\n'は改行を表す文字
print('\n')

# 1行ずつ出力
for line in content:
    print(line)
```

write()メソッド

File オブジェクトの write() メソッドを使うと、ファイルに書き込むことができます。この時、 write() メソッドは書き込まれた文字数を返します。ただし、読み込みモードで開いたファイルに書き込むことはできません。ファイルに書き込むには、書き込みモード、もしくは、追記モードでファイルを開く必要があります。

open() 関数の第2引数に 'w' を渡すと書き込みモードでファイルを開きます。書き込みモードでは、既存のファイルの内容を上書きして書き直します。

```
data_file = open('data.txt', 'w')
```

open() 関数の第2引数に 'a' を渡すと追記モードでファイルを開きます。追記モードでは、既存のファイルの終端に追加して書き込みます。

```
data_file = open('data.txt', 'a')
```

書き込みモードでも追記モードでも、もし引数で指定したファイルが存在しなければ、新たに空のファイルが引数で示したパスに作成されます。

ファイルを読み書きした後は、 close() メソッドを呼び出して閉じます。

In [ ]:
```
myfile = open('greet.txt', 'w')
myfile.write('Hello\n')
myfile.close()

myfile = open('greet.txt', 'a')
myfile.write('Bonjour\n')
myfile.close()

myfile = open('greet.txt')
content=myfile.read()
print(content)
myfile.close()
```

モジュール

Pythonでは、これまでに見てきたような `print` , `len` , `range` などの組み込み関数と呼ばれる基本的な関数を使用することができます。さらに、Pythonには標準ライブラリと呼ばれるモジュール群も含まれています。

各モジュールは関連する関数を備えたPythonプログラムで、それらの関数を利用し独自のPythonプログラムを作成することができます。例えば、 `math` モジュールには、数学関連の関数が含まれています。

モジュールの中の関数を呼び出すには、まず `import` 文を使ってモジュールを読み込む必要があります。

```
import モジュール名
```

モジュールを読み込んだら、その中の関数を使うことができます。以下では、 `math` モジュールの `sqrt` 関数（平方根を計算する関数）を `math.sqrt` と指定して使用しています。

In [ ]:
```
import math
print(math.sqrt(4))
```

`import` は、以下のように `from` と組み合わせて書くこともできます。この時は、モジュールの関数を使う際にモジュール名を書く必要はありません。

```
from モジュール名 import *
```

In [ ]:
```
from math import * #  mathモジュールのsqrt関数だけ使う場合は、from math impo
print(sqrt(4))
```

CSVファイル

csvモジュール

CSV（Comma Separated Values）ファイルは、プレーンテキストファイルに記録された簡易的なスプレッドシート（エクセルの表など）です。CSVファイルの各行は、スプレッドシートの行を表していて、区切り文字（通常はカンマ）は、その行の要素を区切るものです。

```
## 成績CSVファイル
#  ユーザID, 国語、算数
1, 60, 70
2, 70, 90
3, 80, 80
...
```

csvモジュールを使うとCSVファイルを読み書き、解析することが容易になります。csvモジュールを使って、CSVファイルからデータを読み込むには、まず通常のテキストファイルと同様に、 `open()` 関数でCSVファイルを開きます。この `open()` 関数の戻り値の `File` オブジェクトを、 `csv.reader()` 関数へ渡します。すると、 `csv.reader()` は、 `Reader` オブジェクトを生成し、この `Reader` オブジェクトを用いると、CSVファイルの行を順番に処理することができます。

`Reader` オブジェクトを `list()` 関数に渡すと、元のcsvファイルの1行がリストとそのリスト要素としたリスト（リストのリスト）が返ってきます。このリストに対して、リストのインデックスを使うことで、特定の行と列の値に以下のようにアクセスすることができます。

In [ ]:
```
import csv
score = open('simple_score.csv')
score_reader = csv.reader(score)
score_data = list(score_reader)
print(score_data)

# 2行目の3列目
print(score_data[1][2])

# 3行目の4列目
print(score_data[2][3])
```

Reader オブジェクトからは、以下のように for 文をつかうことで1行ずつファイルを処理することもできます。この時、各行はリストとなって取り出されます。以下では、行番号を取得するのに、 Reader オブジェクトの line_num 変数を使っています。 Reader オブジェクトは、1度繰り返し処理すると、再び利用することはできないため、元のCSVファイルを読み込み直すには、ファイルを開き直す必要があります。

In [ ]:
```
import csv
score = open('simple_score.csv')
score_reader = csv.reader(score)
for row in score_reader:
    print(str(score_reader.line_num)+": "+str(row))
```

Writer オブジェクトを用いると、データをCSVファイルに書き込むことができます。 Writer オブジェクトを作るには csv.writer() 関数を使います。CSVファイルにデータを書き込むには、まず、 open() 関数に 'w' を渡して書き込みモードでファイルを開きます。この File オブジェクトを、 csv.writer() 関数に渡して Writer オブジェクトを生成します。

Writer オブジェクトの writerow() メソッドは、引数にリストをとります。この引数に渡されたリストの各要素の値が、出力するCSVファイルの各セルの値となります。

In [ ]:
```
import csv
score = open('my_score.csv', 'w')
# score = open('my_score.csv', 'w', newline='') #for Windows
score_writer = csv.writer(score)
score_writer.writerow(['user','kokugo','sugaku'])
score_writer.writerow([1,50,50])
score_writer.writerow([2,30,40])
score.close()
```

カンマの代わりにタブ区切りで出力したい場合は、以下のように csv.writer() 関数の引数 delimiter に区切り文字を指定します。

```
score_writer = csv.writer(score, delimiter='\t')
```

エラーが出たら

プログラムが実行できない（エラーが出た）時は、バグを取り除くデバッギングが必要になります。例えば、以下に留意することでバグを防ぐことができます。

- "よい"コードを書く
  - コードに説明のコメントを入れる
  - 1行の文字数、インデント、空白などのフォーマットに気をつける
  - 変数や関数の名前を適当につけない
  - グローバル変数に留意する
  - コードに固有の"マジックナンバー"を使わず、変数を使う
  - コード内でのコピーアンドペーストを避ける
  - コード内の不要な処理は削除する
  - コードの冗長性を減らすようにする など
  - 参考
    - Google Python Style Guide (http://works.surgo.jp/translation/pyguide.html)
    - Official Style Guide for Python Code (http://pep8-ja.readthedocs.io/ja/latest/)
- 関数の単体テストを行う
- 一つの関数には一つの機能・タスクを持たせるようにする など

エラーには大きく分けて、文法エラー、実行エラーがあります。以下、それぞれのエラーについて対処法を説明します。

文法エラー

1. まず、エラーメッセージを確認しましょう
2. エラーメッセージの最終行を見て、それがSyntaxErrorであることを確認しましょう
3. エラーとなっているコードの行数を確認しましょう
4. そして、当該行付近のコードを注意深く確認しましょう

よくある文法エラーの例：

- クオーテーションや括弧の閉じ忘れ
- コロンのつけ忘れ
- =と==の混同
- インデントの誤り など

In [ ]:
```
print('This is the error)
```

実行エラー

1. まず、エラーメッセージを確認しましょう
2. エラーメッセージの最終行を見て、そのエラーのタイプを確認しましょう
3. エラーとなっているコードの行数を確認しましょう
4. そして、当該行付近のコードについて、どの部分が実行エラーのタイプに関係しているか確認しましょう。もし複数の原因がありそうであれば、行を分割、改行して再度実行し、エラーを確認しましょう
5. 原因がわからない場合は、print文を挿入して処理の入出力の内容を確認しましょう

よくある実行エラーの例：

- 文字列やリストの要素エラー
- 変数名・関数名の打ち間違え
- 無限の繰り返し
- 型と処理の不整合
- ゼロ分割
- ファイルの入出力誤り など

In [ ]:
```
print(1/0)
```