ダウンロード・データあります

佐藤 聖

頻出の文法を厳選して紹介

● 自己学習の土台作りに

入門者向けの情報がコンパクトにまとめられているPython公式サイトにチュートリアル(1)があります．コンパクトと言っても，項目だけ見てもかなりの量があります．

この中から全ての解説は行いませんが，Pythonの組み込み関数の一覧を**表1**(pp.26-27)に示します．組み込み関数とは，Built-in Functionsのことで，ライブラリをimportしなくても，さまざまな処理を行えます．

本記事では，さらに必要最小限と思われる内容に絞り込み，以下の内容を扱います．

▶標準ライブラリ

import文，int関数，float関数，bool関数，str関数，len関数，range関数，list関数，sort関数，sorted関数

▶式

演算子の優先順位，トークン，デリミタ，代数演算子，比較演算子，ブール演算子，文字列，リスト，タプル，辞書

▶複合文

if文，for文，try文，while文，break文，def文，return文

他の言語で学習経験がある方には少なすぎると感じられるかもしれません．さまざまな種類のプログラムを開発するときには，これだけでは不足することがあると思います．しかし，最も利用頻度の高い内容に厳選しておりますので，最初の一歩を踏み出して，自己学習できる知識の土台を作るには足りると考えました．

● 詳しく知りたくなったら…

Pythonは入門者にとって学習しやすいコンピュータ言語と言われますが，極めようとするとPythonの細部の理解が必要になり，なかなか難しいです．

さらに詳しく知りたい方のために，Python公式サイトにあるドキュメントを紹介します．入門からステップアップする際に参考になります．

▶Python 言語リファレンス(2)

Pythonの仕組みや機能を理解する上で参考にできます．文句解釈，データ・モデル，実行モデル，インポート・システム，式，単純文，複合文などについて書かれています．

各項目の厳密な構文とセマンティクス注1について説明されています．これを読むとPythonでプログラムを実行したときに，どのような動作になるか理解できます．

▶Python標準ライブラリ(3)

必須でない組み込みオブジェクト型や組み込み関数，組み込みモジュールに関するセマンティクスなどの記載もあります．これを参照する場面は，Pythonの異なるバージョンで関数の動作の違いや引数の仕様を確認するときなどでしょう．Pythonのソースコードを読んで実装を確認するよりも簡単なはずです．

◆参考文献◆

(1) Python公式サイトのチュートリアル．
https://docs.python.org/ja/3.7/tutorial/index.html#tutorial-index
(2) Python 言語リファレンス．
https://docs.python.org/ja/3.7/reference/index.html
(3) Python標準ライブラリ．
https://docs.python.org/ja/3.7/library/index.html#library-index

注1：コードの意味，データの形式や構造などを含むソースコード中で利用されている変数や文が正しく動作するかを判断する基準．

1-1 単純な変数

図1 変数はオブジェクトを入れておく入れ物

● 変数の利用目的

Pythonは事前に変数を宣言せず，変数で扱うオブジェクト型を指定しなくても利用できます．これはPythonの学習しやすい特徴の1つです．変数に値を代入する際に，データの型は自動的に定義されます．しかし，オブジェクト型を意識せずに利用できるため関数の引数として利用するとき，変数に格納されているオブジェクトの型が適切でなくエラーになってしまうことがあります．変数の利用イメージを**図1**に示します．

リスト1 Pythonの予約語や標準ライブラリに含まれる文字列

```
False, await, else, import, pass, None, break, except,
    in, raise, True, class, finally, is, return, and,
  continue, for, lambda, try, as, def, from, nonlocal,
while, assert, del, global, not, with, async, elif, if,
                                        or, yieldなど
```

● 変数に値を代入する書式

代入する書式は次になります．

```
a = 整数
a = 小数
a = 文字列
a = 変数
```

変数名は，英数字とアンダスコア(アンダバー)の組み合わせで指定します．Python 3は漢字も利用できますが，WindowsとLinuxなど異なるOSでは文字コードの違いでうまく機能しないことがあるかもしれません．お勧めなのは英数字とアンダスコアの組み合わせで作る変数名です．ただし，変数名の1文字目に数字を使うことはできません．変数名は大文字と小文字が区別されます．

さらにPythonの構文などで利用される予約語や標準ライブラリに含まれる文字列(**リスト1**)と，明示的な宣言で呼び出したPython拡張ライブラリやパッケージに含まれるライブラリ名，モジュール名，クラス名，メソッド名(関数名)と同じ文字列を変数名に指定できません．変数名を決めるときには，他の要素と重ならないユニークな文字列になるよう工夫することが必要です．

変数aにオブジェクトを代入するには，

```
a =
```

の後に，数値や文字列，他の変数の値，リスト，タプルや辞書などの値を指定します．変数にオブジェクトを代入する際に半角スペースを入れずに，

```
a=値
```

としても，

```
a = 値
```

としても同じようにオブジェクトが代入されます．

▶型を明示しなくても使える

変数に入れる値には，整数や小数付きの値，文字列などがあり，これらの種類を型と言います．

例えば整数を入れた変数であれば，その変数の型は整数ということになります．

C言語やJava言語などプログラムの実行にコンパイルが必要な言語の多くには，異なる型の値を代入する際に，変数の型を指定する手続きが必要です．

しかしPythonは変数にオブジェクトを代入するときにオブジェクトの型が自動的に定義されます．また，変数に異なるオブジェクトを代入すると，そのオブジェクトの型に合わせて自動的に再定義されます．プログラムを作成するとき，多数の変数を扱いますが，この機能のおかげで短いソースコードを記述するだけで済みます．

また，他のコンピュータ言語では意味が異なることもありますが，Pythonでは変数名を'(シングル・クォート)や"(ダブル・クォート)のいずれかで囲うことで変数として機能しなくなり，変数名の文字列として扱われます．

● 実行例

▶整数

図2は，変数aにオブジェクトの数値5を代入して，print関数で画面出力した例です．変数にオブジェクトを代入する前に変数を初期化したり，代入時にオブジェクト型を指定したりしなくても自動的に定義されます．念のため，type関数で変数aのオブジェクト型を確認します．<type 'int'>が出力されていますが，integralの省略形のint型となっており，オブジェクト型は整数であることが分かります．

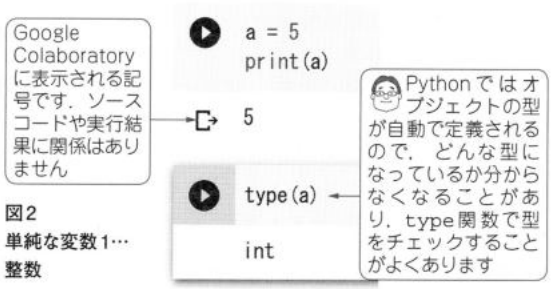


図2
単純な変数1…整数

▶小数

図3は，整数ではなく，小数点以下を含む値5.0を変数aに代入する例です．print関数で変数aのオブジェクトを画面出力すると，5.0が表示されました．結果は，オブジェクトの型が自動的に定義されるので整数の5にならず，5.0として保持されています．また，type関数で変数aのオブジェクト型を確認するとfloat型でした．

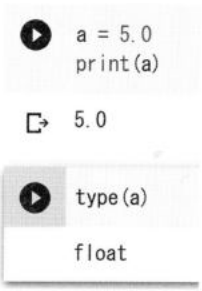


図3
単純な変数2…小数

▶文字列

変数aにオブジェクトとして文字列を代入するときは，文字列の前後を'や"で囲います．

注意点としては，文字列の前に'を付けたら，文字列の後も同じ'を付けることが必要です．前後で同じクォーテーション('もしくは")を使用しないと文字列としてオブジェクト型が自動的に定義されません．

図4のようにprint関数で変数aのオブジェクト

を表示すると，文字列のみが画面出力されます．type関数で変数aのオブジェクト型を確認するとstr型になっています．

```
a = "Hello World"
print(a)
```

```
Hello World
```

```
type(a)
```

```
str
```

図4
単純な変数3…文字列

▶変数同士の代入

変数に格納済みのオブジェクトを他の変数に代入できます．**図5**では変数aのオブジェクトの5を変数bに代入しています．変数に数値や文字列のオブジェクトを代入するのと同様に，変数間でオブジェクトを代入するときにも=が使えます．print関数の引数に変数bを渡すと，画面には5が出力されます．また，type関数で変数aと変数bのオブジェクト型を確認すると，いずれもint型になっています．変数間の代入でオブジェクトだけでなくオブジェクト型も自動的に定義されており，オブジェクト型の宣言せずに代入できていることが分かります．

Pythonでは同じ変数名を異なる型のオブジェクトを代入するのに何度も再利用すると，変数に代入されたオブジェクトの型を見失いやすくなります．

例えば，変数aに整数を代入して利用した後に，別の処理で変数aに文字列を代入するような使い方も可能です．しかし，ソースコードのどの行でオブジェクト型が変わったのか分かりづらくなります．

型によって使用する変数を分けて，どんなオブジェクトが代入されるのか変数名を見れば分かるようにするとよいです．

```
[1]  a = 5
     b = a
     print(b)
```

変数同士で代入するときにはオブジェクトだけではなく，型も引き継がれます．型を変えたいときには関数や書式を使います

```
     5
```

```
[2]  type(a)
```

```
     int
```

```
type(b)
```

```
int
```

図5
単純な変数4…変数同士の代入

1-2　複数の値をまとめて扱う変数

● 変数の利用目的

変数を使うときに1つの値だけでなく，複数の値をまとめて扱いたいことがあります．Pythonでは複数の値を扱う方法が幾つかあります．詳しくは後述しますが，ここでは複数の値をまとめて扱う方法があるということを覚えておいてください．

● 複数の値を扱う変数の書式

代入の書式は次です．

```
a = リスト([]で囲われたオブジェクトの列)
a = タプル(()で囲われたオブジェクトの列)
a = 辞書(ディクショナリ，{}で囲われたキーとオブジェクトの列)
'a'　変数名を表す文字列．囲われた部分が文字列として扱われる
"a"　囲われた部分が文字列として扱われる
```

● 実行例

▶辞書

辞書は，{と}で囲われたキーとオブジェクトの列です．**図6**では，変数aのオブジェクトとして代入しています．辞書のキーはオブジェクトを参照する際に利用します．リストやタプルでは要素の順序に意味を持っていますが，辞書では要素の順序は一定しておらず，連想配列やハッシュなどとも呼ばれます．そのため，要素の位置でオブジェクトを参照したり，更新したりできませんので，キーを使ってオブジェクトを操作します．type関数で確認すると変数aのオブジェクトはdict型に定義されています．

辞書の詳しい利用方法は後述します．

```
a = {'A':1,'B':'apple'}
print(a)
```

```
{'A': 1, 'B': 'apple'}
```

```
type(a)
```

```
dict
```

図6
複数の値をまとめて扱う変数1…辞書

▶変数の内容の表示

図7のように，変数aの変数名をディスプレイ表示する場合，'や"で囲んでprint関数の引数にすると，いずれも変数名が表示されます．

type関数で確認すると変数aのオブジェクトはstr型になっており，文字列として定義されています．このように変数や文字列としての扱いが異なります．

```
[1]  a = 5
     print(a)

     5

[2]  type('a')

     str

[3]  print("a")

     a
```

変数として利用するときには，そのままの文字で記述します．'や"で囲うと変数として機能しません

図7
複数の値をまとめて扱う変数2
…変数の内容の表示

```
type("a")

str
```

▶リスト

リストは[と]で囲まれたオブジェクトの列です．図8では，このリストを変数aのオブジェクトとして代入します．以下の例では，1，2，3を[1，2，3]のように記述してリストを定義し，このリストを変数aのオブジェクトとして代入しています．変数へ代入するときには，数字や文字列のときのように=を使います．print関数で画面出力すると，リスト内のオブジェクトがそのまま表示されます．

また，type関数で確認すると変数aのオブジェクトはlist型です．リストの詳しい利用方法は後述します．

```
a = [1,2,3]
print(a)

[1, 2, 3]
```

図8
複数の値をまとめて扱う変数3
…リスト

```
type(a)

list
```

▶タプル

タプルは(と)で囲われたオブジェクトの列です．図9でも変数aのオブジェクトとして代入します．タプルはリストと異なり，オブジェクトの追加・更新・削除ができません．定義したタプルに格納されているオブジェクトを参照するために利用します．

変数aにタプルを代入してprint関数で出力すると，1，2，3が格納されたタプルが画面出力されます．また，type関数で確認すると変数aのオブジェクトはtuple型に定義されています．タプルの詳しい利用方法は後述します．

```
a = (1,2,3)
print(a)

(1, 2, 3)
```

図9
複数の値をまとめて扱う変数4
…タプル

```
type(a)

tuple
```

▶リストとタプルの違い

リストとタプルの両者は似ていますが，できることが異なります．[と]で作るリストはオブジェクトの追加・変更・削除が可能なのに対し，(と)で作るタプルは追加・変更・削除が不可能で参照するだけです．

タプルはプログラムの中で変更しない，または変更されては不都合なオブジェクトを格納するのに使われます．

図10のようにリストもタプルも要素として保持しているオブジェクトを変数に代入して，print関数で画面出力できます．この使い方では処理の振る舞いに違いはありません．

```
a = ['c', 'a', 'b']
print(a)

['c', 'a', 'b']

a = ('c', 'a', 'b')
print(a)

('c', 'a', 'b')
```

よく見ないと違いが分かりづらいので，オブジェクトの使い道を理解して，型をチェックしましょう

図10
一見するとリストとタプルの違いは分かりにくい

リストとタプルの動作の違いを確認するために，リストのオブジェクトを並び変えてみます．sort関数を使用すると図11のように変数aに格納されたリストのオブジェクトを並び替えることができます．その後，print関数の引数に変数aを渡すとオブジェクトが昇順に並び変えられたリストが表示されます．リスト内の要素は，オブジェクトとともに位置が変更されていますので，元のオブジェクトの順序は保存されていません．

```
a = ['c', 'a', 'b']
a.sort()
print(a)
```

昇順の並び替えでは，変数名の後に，sort関数名を記述します．こうすると変数名が目立つので読みやすくなります

```
['a', 'b', 'c']
```

図11　リストとタプルの違い1…リスト内のオブジェクトをsort関数で並び替えた結果

タプルはオブジェクトの参照しかできず，sort関数でオブジェクトを並び変えようとすると図12のようなエラーが表示されます．sort関数はタプルのオブジェクトに対して，要素番号を変更する操作をしようとしますが，タプル型(tuple)はそれを受け付けられません．オブジェクトを書き換えなくても要素位置が変わるような操作もタプルでは行えません．タプルを定義した際の要素位置は固定化されていることが分かります．

```
a = ('c', 'a', 'b')
a.sort()

---------------------------------------------
AttributeError
<ipython-input-16-26ec47e5587a> in <module>
      1 a = ('c', 'a', 'b')
----> 2 a.sort()

AttributeError: 'tuple' object has no attr

SEARCH STACK OVERFLOW
```

エラー・メッセージ

◀図12
リストとタプルの違い2…タプル内のオブジェクトをsort関数で並び替えた結果

▶タプルからリストへの変換

タプルはlist関数の引数に渡すことでリスト型に変換できます．リストならばオブジェクトの並び替えが可能になります．

図13では，変数aにタプルを代入して，list関数でリストに変換したものを変数bに代入します．変数bにはリストが格納されているため，sort関数でオブジェクトを並び替えできます．リストに変換した後，print関数で並び替え済みのオブジェクトを確認できます．

タプルに格納されたオブジェクトやそれらの並び順を保持したまま，これらの情報に変更を加えるときにはリストに変換すると扱いやすいです．

```
a = ('c', 'a', 'b')
b = list(a)
b.sort()
print(b)
```

オブジェクトの型を変えるときには，明示的に関数で変換させます

```
['a', 'b', 'c']
```

図13　タプルからリストへの変換

1-3　変数や文字列を画面に出力するprint関数

●print関数の利用目的

Pythonの学び始めでは，print関数(メソッド)が最も利用される関数の1つです．数字や文字列などを画面表示するときに利用します(図14)．Python標準モジュールの組み込み関数ですのでimport文でライブラリを読み込まなくても利用できます．関数の書式は以下の通りです．

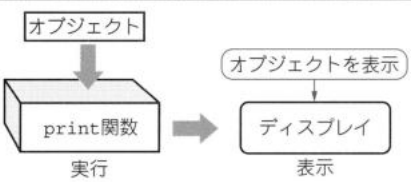


図14　画面に数字や文字列を表示する

●print関数の書式

print関数の書式は次です．

```
print(オブジェクト, sep=' ',
end='¥n', file=sys.stdout,
                    flush=False)
```

関数の引数としてオブジェクト(値)などを設定します．もし，オブジェクトが設定されていないとエラーになります．関数に与えられたオブジェクトは文字列として出力します．変数の項目でも説明しましたが，文字列は変数名と区別するために文字列の前後を'や"で囲います．

▶数字を出力する場合

オブジェクトとして数字を指定した場合，print関数を実行する際にPythonの組み込み関数のstr関

リスト2　文字列と同一行にプロンプト（>>>）が表示される

```
>>> print("a","b","c","d","e","f", sep=",")
a,b,c,d,e,f
>>>                                    ← 改行されている

>>> print('09','12', sep='-', end='¥t')
09-12    >>>                           ← 改行されていない
```

数が利用され，画面表示のためだけに文字列に自動変換されます．

str関数は与えられたオブジェクトを文字列に変換します．明示的にprint(str(数値))のように記述しなくても，print(数値)とするだけで同じように機能し，画面には文字列の数字を出力します．

▶引数

通常，利用頻度は低いですが，print関数の引数としてsep，end，file，flushを指定できます．使い方を工夫すると便利に使える場合がありますので紹介します．

- **sep**

オブジェクトの区切りの文字を設定する引数です．

- **end**

文字列の最後に付加される文字を設定する引数で，デフォルトではエスケープ・シーケンス[注1]の改行(¥n)が設定されており，文字列を出力後に改行が付加されます．この引数を変更することで文字列の後にエスケープ・シーケンス，数値や文字列などのオブジェクトを加えることができます．

- **file**

デフォルトはシステムの標準出力(sys.stdout)である画面出力になっていますが，オブジェクトの出力先をファイルなどに変えられます．

- **flush**

出力がバッファ化するか否かの設定です．デフォルトの設定ではFalseになっており有効になっています．Trueならバッファが無効になります．

● 実行例

図15は，print関数で文字列を画面に表示する例です．"で文字列の前後を囲い，引数として指定します．文字列は'で囲っても同じように処理されます．

▶文字列

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

"や'で囲っている範囲に気を付けましょう．前後のどちらかを付け忘れるとエラーになります

図15　print関数1…文字列

▶数字

図16はprint関数で数値を表示する例です．引数に数字を指定しています．

```
print(1234567890)
1234567890
```

図16　print関数2…数字

▶文字列の連結

図17の例ではprint関数で区切り文字を指定して，2つの文字列を連結して画面出力します．ここでは2つの文字列は'で前後を囲います．

引数sepに区切り文字である@を'で囲んで指定します．各引数は,(カンマ)で区切ります．print関数で区切り文字を指定すると文字列の連結が行えます．

```
print('Hello','World.com', sep='@')
Hello@World.com
```

決まった文字列のパターンがあるなら引数のsepを使うと，異なる部分のみ文字列を指定すればよいです

図17　print関数3…文字列の連結

▶多数の文字列を連結

図18のようにprint関数に多数の文字列を指定して，区切り文字で連結することもできます．

引数sepを使って，複数のオブジェクトをまとめて1行に画面出力するときに便利です．

```
print("a","b","c","d","e","f",sep=",")
a,b,c,d,e,f
```

図18　print関数4…多数の文字列を連結

▶改行の制御

次は，2つの文字列の区切り文字に"-"を，endに"¥t"を設定して表示させる例です．文字列を表示後，その行を改行で終わるのではなく，タブを挿入しています．

Thonny Python IDEやGoogle Colaboratoryで試した場合は，違いが分かりませんが，コマンド・ラインのPythonインタプリタで試すと結果に違いがあります．通常print関数実行後は，改行されて次の行にプロンプト(>>>)が表示されますが，文字列と同一行にプロンプトが表示されます(**リスト2**)．

注1：エスケープ・シーケンスとは，画面上に文字を出力する際に，文字色の変更，カーソルの移動などの文字出力の制御を行う特殊な文字列です．通常は，"¥"と1つの文字を組み合わせた2文字で表現されます．

1-4　モジュールや関数を呼び出すimport文

● import文の利用目的

Python標準ライブラリに含まれる組み込み関数は明示的にライブラリを読み込まなくても利用できます．Pythonの組み込み関数以外の外部のライブラリやパッケージに含まれるモジュールや関数を利用するときにimport文を使って呼び出します．

外部のライブラリやパッケージは，Pythonで記述されたものとは限りません．C/C++などで記述されたものもあり，ライブラリやパッケージの実装によって構造が異なることもあるかもしれません．一般的にはモジュールは関数やクラスなどをまとめて記述した.pyファイルです．そして，モジュールを集めてまとめたものをパッケージと呼びます(図19)．

パッケージには，複数クラスを定義することもでき，整理するのに__init__.pyにクラスを記述してimport文でクラスを呼び出せるようにします．

特定のクラスを呼び出すとき__init__.pyの名前空間を参照します．

パッケージをまとめたものがライブラリです．一般的に呼び方には厳密な定義がないので，おのおののライブラリやパッケージによって表現が異なるかもしれません．

使うときは，以下のような書式で関数を利用できるようソースコードに記述します．*のようにワイルドカードを利用すると，そのモジュールに含まれる全ての関数を呼び出せます．

呼び出し方法には複数あり，モジュールの設計によってどれを使うかが変わります．詳しくは外部ライブラリの説明を読むか，モジュールのソースコードを確認するとよいでしょう．

なお，Pythonの公式ドキュメントでは関数とメソッドという表記が混在していますが，ここでは関数という表記で統一します．

● import文の書式

import文の書式は次です．

• モジュールを呼び出す

```
import モジュール名
```

• モジュールを別名で呼び出す

```
import モジュール名 as 別名
```

• ライブラリやクラス中の関数を直接呼び出す

```
from モジュール名 import クラス名または関数名
from モジュール名 import *
from ライブラリ名.モジュール名 import クラス名または関数名
```

（ライブラリの名前空間を持つ）

import文でモジュール名.クラス名または関数名を呼び出します．

このように宣言することで他のファイルに記述された関数を利用できます．

もし，全てのプログラムを1つのファイルにまとめて記述するとソースコードが長くなり可読性が悪くなります．複数のファイルに分割して記述することで，可読性をよくし，プログラムの再利用性も高くなります．

▶モジュールに使いやすい名前を割り当てられる

短い別名で呼び出すことができます．例えば，numpyとして記述すべきところを，短い別名としてnpを定義して呼び出すといったように利用します．この場合ソースコード内でnumpyの機能を利用するときには，“numpy.関数名”のように記述せずに，“np.関数名”で利用することが可能になります．

別名を使うことでクラス名.関数名のように書くよりもソースコードが短くなります．モジュールの実装によっても異なりますが，この仕組みを使わないと次のように記述が長くなってしまいます．

```
モジュール名.クラス名.関数名
ライブラリ名.モジュール名.クラス名.関数名
```

関数名，変数名やそのほか予約文字列と同一でないユニークな文字列であれば別名として定義できます．

numpy以外にも，pandasライブラリならpd，beautifulsoupライブラリならbsなど，一般的によく利用される別名があります．

● 実行例

▶timeモジュールの呼び出し

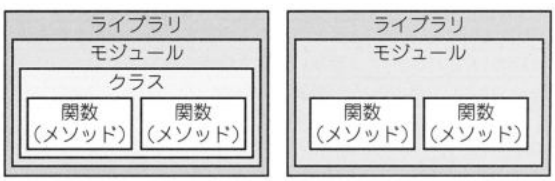
import文を使って時刻を扱う関数，処理の一時停止などに利用される機能を持つtimeモジュールを呼び出す例です．このモジュールに含まれる関数を利用する場合，

```
time.関数名
```

のように記述して利用します．関数に引数を設定できる場合，

```
time.関数名(引数)
```

のようにカッコ内に記述します．次のように使います．



(a) クラスのある場合　(b) クラスのない場合

図19　ライブラリ，モジュール，クラス，関数の関係

```
import time
```

▶Minecraftライブラリの呼び出し

特設記事で利用するMinecraftのAPI操作に，mcpiライブラリを利用します．このライブラリに別名としてmcを割り当て呼び出す例です．

import文での呼び出しが長い文字列になると，記述ミスをしやすいですし，何度も関数を利用するときにソースコードが冗長になることで読みにくくなります．別名を利用すれば，ソースコードを簡潔に記述でき，読みやすくなります．

```
import mcpi as mc
```

mcpiライブラリにminecraftクラスがあり，さらにその配下にMinecraft関数が含まれます．import文でmcpiライブラリを呼び出して関数を利用する際にはクラス名も記述するため，

```
mcpi.minecraft.Minecraft.関数名
```

のようになります．

関数の引数やサブ関数を記述すると，さらに長いソースコードになります．これだと非常に冗長なソースコードになるので，import文でmcpi.minecraftに含まれるMinecraftクラスを呼び出す方が，読みやすいソースコードです．

```
from mcpi.minecraft import Minecraft
```

Minecraftクラスに含まれる関数の利用は，

```
Minecraft.関数名
```

のように記述して利用します．

ライブラリやパッケージの構造によっては，このように呼び出すことができないこともあります．ライブラリやパッケージのリファレンスや付属するReadmeファイルで利用方法を確認してください．

▶datetimeライブラリの使用例

import文で日付型を扱うdatetimeライブラリを読み出すには，**図20**の1行目のように記述します．

import文で呼び出したクラスに含まれるクラスや関数を利用する例は2行目にあります．

datetimeライブラリからdateクラスのtoday関数を使って今日の日付を取得し，print関数で画面出力しています．

type関数でdatetime.date.today()が返すオブジェクトの型を調べるとdatetime.date型となっており日付であることが分かります．

図20　import文…datetimeライブラリの使用例

さとう・せい

表1　Pythonの組み込み関数

関数名	説　明
abs()	引数は整数または浮動小数点数を与え数の絶対値を返す
all()	イテラブルの全ての要素が真ならば(もしくはイテラブルが空ならば) Trueを返す
any()	イテラブルのいずれかの要素が真ならばTrueを返し，イテラブルが空ならFalseを返す
ascii()	オブジェクトの印字可能な表現を含む文字列を返す
bin()	整数を先頭に0bが付いた2進文字列に変換する
bool()	引数は標準の真理値判定手続きを用いて変換され，TrueまたはFalseのどちらかを返す
breakpoint()	呼び出された箇所からデバッガへ移行する
bytearray()	0 ≦ x < 256の範囲の整数からなる新しいバイト配列を返す
bytes()	範囲0 ≦ x < 256の整数のイミュータブルなシーケンスであるbytesオブジェクトを返す
callable()	引数が呼び出し可能オブジェクトであればTrueを，そうでなければFalseを返す
chr()	引数のオブジェクトがユニコード・ポイントが整数である文字を表す文字列を返す
classmethod()	メソッドからクラス・メソッドへ変換する
compile()	引数のsourceをコード・オブジェクト，もしくはASTオブジェクトにコンパイルする
complex()	値real＋値imagの複素数を返すか，文字列や数を複素数に変換する
delattr()	オブジェクトが許すなら，指名された属性を削除する
dict()	新しい辞書を作成する
dir()	引数がない場合，現在のローカル・スコープにある名前のリストを返す．引数がある場合，そのオブジェクトの有効な属性のリストを返そうと試みる
divmod()	2つの(複素数でない)数を引数として取り，整数の除法を行ったときの商と剰余からなる対を返す
enumerate()	enumerateオブジェクトを返す
eval()	引数をPythonの式として構文解析し評価する
exec()	オブジェクトがコード・オブジェクトなら実行される．文字列ならPython文として解析され，実行される

表1　Pythonの組み込み関数（つづき）

関数名	説　明
filter()	イテラブルの要素のうちfunctionが真を返すものでイテレータを構築する
float()	オブジェクトは数または文字列から生成された小数を返す
format()	引数の値と書式で書式化された表現に変換する
frozenset()	新しいfrozensetオブジェクト（イミュータブル名オブジェクトで辞書のキーや他の集合の要素として用いることができる）を返す
getattr()	オブジェクトに指名された属性の値を返す
globals()	現在のグローバル・シンボル・テーブルを表す辞書を返す
hasattr()	引数はオブジェクトと文字列で，文字列がオブジェクトの属性名の1つであった場合Trueを，そうでない場合Falseを返す
hash()	オブジェクトから整数のハッシュ値を返す
help()	組み込みヘルプ・システムを起動する
hex()	引数の整数を0xで始まる小文字の16進文字列に変換する
id()	オブジェクトの識別値を返す
input()	文字列の引数が存在すれば末尾の改行を除いて，標準出力に書き出す
int()	数値または文字列から作成された整数オブジェクトを返す
isinstance()	オブジェクト引数がclassinfo引数のインスタンスであるか，サブクラスのインスタンスの場合にTrueを返す．そうでない場合はFalseを返す
issubclass()	クラスがclassinfoのサブクラスである場合にTrueを返す
iter()	イテレータ・オブジェクトを返す
len()	オブジェクトの長さ（要素の数）を返す
list()	イテラブルの項目からリストを構築する
locals()	現在のローカル・シンボル・テーブルを表す辞書を更新して返す
map()	反復可能オブジェクト（リスト，タプルなど）の各項目に設定された関数を適用した後，イテレータのマップ・オブジェクトを返す
max()	イテラブルの中で最大の要素，または2つ以上の引数の中で最大のものを返す
memoryview()	与えられたオブジェクトから作られたメモリ・ビュー・オブジェクトを返す
min()	イテラブルの中で最小の要素，または2つ以上の引数の中で最小のものを返す
next()	イテラブルの__next__()メソッドを呼び出すことによってこのオブジェクトの次の要素を取得する
object()	特徴を持たない新しいオブジェクトを返す
oct()	整数を先頭に0oが付いた8進文字列に変換する
open()	ファイルを開き，対応する ファイル・オブジェクトを返す
ord()	1文字のユニコード文字を表す文字列に対し，その文字のユニコード・ポイントを表す整数を返す
pow()	引数にx，yおよびzを代入できる．xのy乗を返す．zがあれば，xのy乗に対するzの剰余を返す
print()	オブジェクトをsepで区切りながらテキスト・ストリームfileに表示し，最後にendを表示する
property()	property属性を返す
range()	イミュータブルなシーケンス型を返す
repr()	オブジェクトの印字可能な表現を含む文字列を返す
reversed()	要素を逆順に取り出すイテレータを返す
round()	数字の小数部を引数のndigists桁に丸めた値を返す．ndigitsを省略，Noneの場合，入力値に最も近い整数を返す
set()	オプションでイテラブルの要素を持つ新しいsetオブジェクトを返す
setattr()	引数はオブジェクトに対し，文字列で与えられた既存の属性または新たな属性の名前を付与する
slice()	range(start, stop, step)で指定されるインデックスの集合を表す，スライス・オブジェクトを返す
sorted()	イテラブルの要素を並べ替えた新たなリストを返す
staticmethod()	メソッドを静的メソッドへ変換する
str()	オブジェクトのstr型を返す
sum()	開始とイテラブルの要素を左から右へ合計し，総和を返す
super()	メソッドの呼び出しをtypeの親または兄弟クラスに委譲するプロキシ・オブジェクトを返す
tuple()	イテラブルの項目からタプルを構築する
type()	オブジェクトの型を返す
vars()	モジュール，クラス，インスタンス，あるいはそれ以外の__dict__属性を持つオブジェクトの__dict__属性を返す
zip()	イテラブルから要素を集めたイテレータを作る
__import__()	この関数はimport文により呼び出される

※引用および参考：Python公式ウェブ・サイトの「Python標準ライブラリ」より（https://docs.python.org/ja/3.7/library/functions.html）

2-1 整数を扱うint関数

図1 文字列を数値に変換できる

変数に入れる値には整数や少数付きの値，文字列などがあります(前章)．これらの種類を型と言います．例えば，変数に整数を入れた場合は，その変数の型は「整数型」ということになります．

第2章では，その変数型の変換をする関数の説明をします．

● int関数の利用目的

int関数は引数に指定された数値また文字列の数値から10進数の整数を返します．数値には，整数，小数を指定でき，もう1つの引数として基数を指定できます(図1)．

Pythonでは明示的にデータ型を指定しなくても変数やリストに代入できます．

例えば，PythonプログラムでSNSの書き込みやオンライン・ショッピング・サイトのユーザ評価などの情報を収集したときに，文字列の中の数字を使って計算したい場合があります．

しかし，文字情報から抽出された数字は'で囲われた文字列ですので数値として計算できません．+演算子を使うと文字列同士をつなげて文字列結合されてしまいます．

そのようなときにint関数で文字列の数字を整数に変換することで計算できるようにします．

● int関数の書式

int関数の書式は次になります．

```
int(オブジェクト, base=10)
```

引数のオブジェクトには，数値または文字列を指定します．オブジェクトが与えられないと関数から0が返されます．baseは基数の設定になり，この引数を省略するとデフォルトの10進数になります．

baseは0と2～36の範囲で指定できます．しかし，実際に利用されるのは，2，8，10，16のいずれかになり，それぞれ2進数，8進数，10進数，16進数の意味です．

● 実行例

▶10進数として数値に変換

図2は，int関数の引数のオブジェクトに数字の文字列(10)を指定して，print関数で結果を画面出力した例です．引数baseを省略したのでデフォルトの10進数です．画面には整数の10が表示されます．

type関数でint関数が返すオブジェクトの型を確認するとint型ですので，整数であることが分かります．

```
print(int('10'))
10
type(int('10'))
int
```

ウェブ・サイトから取得したテキスト中の数字を数値として計算に利用するときにオブジェクト型に変換することがあります

図2 int関数1…10進数として数値に変換

▶基数を指定して数値に変換

図3の例では，引数のオブジェクトに2進数，8進数，16進数の値を設定して，10進数の10を画面出力しています．10進数以外の基数の場合，baseの値は省略できません．ここでは2つ目の引数を設定するときに

```
base =
```

を省略しました．いずれも画面出力は10進数の10です．

type関数でint('A',16)が返すオブジェクトの型はint型になっており整数です．int関数では整数への変換と基数の変換の両方を扱えます．

```
[2] print(int('1010',2))
10
[3] type(int('1010',2))
int
[4] print(int('12',8))
10
[5] type(int('12',8))
int
[6] print(int('A',16))
10
type(int('A',16))
int
```

Google Colaboratoryで入力する位置に応じて表示される文字です．プログラムや実行結果とは無関係です

10進数と2進数との変換や，10進数と16進数との変換が簡単です
マイコンやセンサから受信したデータを10進数にすれば分かりやすくなります

図3 int関数2…基数を指定して数値に変換

2-2　小数を扱うfloat関数

● float関数の利用目的

float関数は引数に与えられた整数，文字列の整数および小数のオブジェクトを浮動小数点数に変換します(図4)．数値の型を合わせて計算精度を高めるためにオブジェクト型の変換に利用します．

例えば，ウェブ・スクレイピングによって収集される，センサ・データ，金融商品の価格データや天気予報の気温データなどの浮動小数点数で扱われるデータを活用したいときがあります．

データ型を指定せずに変数やリストにデータを代入すると，取得したデータが'で囲われた文字列になることがあります．それらのデータを活用して計算したり，グラフを作成したりする際には文字列のまま利用できません．float関数で文字列の小数を浮動小数点数に変換して計算やグラフ作成に活用します．

● float関数の書式

float関数の書式は次になります．

```
float(オブジェクト)
```

float関数は，数または文字列のオブジェクトを受け取り，浮動小数点数を返します．オブジェクトが半角および全角のアラビア数字で'や"で囲われた文字列の場合，10進数を含んだ文字列の数値として扱われます．

今回は詳しく扱いませんが，NaNや正負の無限大を表すinf，Inf，INFINITY，iNfINityといった文字列も設定できます．

● 実行例

▶整数を浮動小数点数に変換

整数を浮動小数点数に変換する場合は，図5の例のようにfloat関数のオブジェクトに整数を設定します．type関数でfloat関数が返すオブジェクトの型はfloat型になっており，浮動小数点数であることが分かります．

```
print(float(12345))
12345.0
type(float(12345))
float
```

図5　float関数1…整数を浮動小数点数に変換

図4　整数や数値を表す文字列を浮動小数点数に変換する

▶符号付きの場合

図6のように正負の符号付きの整数を浮動小数点数に変換できます．いずれもfloat型のオブジェクトが返されます．

```
[1] print(float(+12345))
12345.0
[2] type(float(+12345))
float
[3] print(float(-12345))
-12345.0
type(float(-12345))
float
```

図6　float関数2…整数を符号付きの浮動小数点数に変換

▶オブジェクトを指定しなかった場合

図7のように0.0が返されます．type関数にオブジェクトを設定しないfloat関数を引数として渡した場合，float型でした．

```
print(float())
0.0
type(float())
float
```

オブジェクトを忘れてもエラーにならないので，思いもよらない処理が実行されないように注意が必要です

図7　float関数3…オブジェクトを指定しなかった場合

▶文字列を指定した場合

図8のようにオブジェクトに文字列の半角および全角の数字を設定しても浮動小数点数に変換されます．また，文字列にエスケープ・シーケンスを含んでいても正常に変換されます．また，文字列の前後に半角スペースが入っていても，数字部分が浮動小数点数に変換されます．

```
[1] print(float("12345"))
    12345.0

[2] type(float("12345"))
    float

[3] print(float(' １２３４５'))
    12345.0

[4] type(float(' １２３４５'))
    float

[5] print(float("12345¥n"))
    12345.0

[6] type(float("12345¥n"))
    float

[7] print(float(" 12345 "))
    12345.0

    type(float(" 12345 "))
    float
```

ウェブ・サイトから取得したデータには，全角や半角の数字が混ざっているので，数値に変換して計算に使えるのは便利です

図8　float関数4…文字列を指定した場合

type関数でオブジェクトの型は，いずれもfloat型になっています．

▶NaNを指定した場合

図9のように文字列のNaNを指定した場合，nanが返されます．このため，テーブルを扱う場合，変数に値を代入したときにNaN(Not a Number)になっていても，エラーにならず処理されます．type関数でもオブジェクトの型はfloat型です．

```
print(float("NaN"))
nan

type(float("NaN"))
float
```

図9　float関数5…NaNを指定した場合

文字列のInfとINFINITYを指定した場合，図10のようになります．

```
type(float("Inf"))
float

type(float("INFINITY"))
float
```

図10　float関数6…Infを指定した場合

2-3　真理値を扱うbool関数

● bool関数の利用目的

bool関数は，数値，文字列やその他の型のオブジェクトをbool型(ブール型)に変換する組み込み関数です(図11)．bool型は整数型のサブクラスです．

if文の条件式，変数やデーブルに含まれるオブジェクトの真偽判定に利用します．

● bool関数の書式

bool関数の書式は次になります．

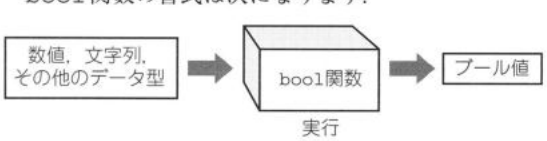


図11　数値や文字列をブール値に変換する

bool(オブジェクト)

この関数は引数のオブジェクトからbool値(TrueまたはFalse)を返します．bool値のインスタンスはFalseとTrueだけです．

オブジェクトは，標準の真理値判定手続きを用いて変換されます．

● Falseと判定されるオブジェクト

偽であると定義されている定数のNone，False以外にも以下のオブジェクトが偽と判定されます．

・数値型のゼロ
　0，0.0，0j，Decimal(0)，Fraction(0，1)
・空のシーケンスまたはコレクション
　''，()，[]，{}，set()，range(0)

オブジェクトが偽(False)または省略(オブジェクトなし)されている場合，この関数はFalseを返します．なお，引数としてこれら以外のオブジェクトを渡すとTrueを返します．

● 実行例

▶数字を指定した場合

bool関数のオブジェクトに数字を指定した場合，図12のようにTrueが返ってきます．type関数でオブジェクトの型を確認するとbool型です．

```
print(bool(12345))
True
type(bool(12345))
bool
```

図12　bool関数1…数字を指定した場合

▶何も指定しない場合

bool関数のオブジェクトを指定しない場合，図13のようにFalseが返ってきます．type関数でbool関数が返すオブジェクトの型はbool型です．

```
print(bool())
False
type(bool())
bool
```

「オブジェクトの有無をチェックするのに使えます」

図13　bool関数2…何も指定しない場合

▶Noneを指定した場合

bool関数のオブジェクトにNoneを代入すると図14のようにFalseが返ってきます．Noneは何もない状態を表すnullオブジェクトです．

```
print(bool(None))
False
type(bool(None))
bool
```

図14　bool関数3…Noneを指定した場合

▶TrueやFalseを指定した場合

図15のようにbool関数のオブジェクトにTrueやFalseを指定した場合，これらのオブジェクトはbool型ですので，type関数でオブジェクトの型を確認すると，そのままの型で返されていることが分かります．

```
[1] print(bool(True))
True
[2] type(True)
bool
[3] type(bool(True))
bool
[4] print(bool(False))
False
[5] type(False)
bool
type(bool(False))
bool
```

「None，True，Falseは，変数名に使うことができません」

図15　bool関数4…TrueやFalseを指定した場合

▶文字列を指定した場合

bool関数は文字列の有無で判定する場合，図16のように文字列をオブジェクトとして代入するとTrue，文字列がないときにはFalseが返ってきます．

```
a = "12345"
print(bool(a))
True
a = ""
print(bool(a))
False
```

「変数にオブジェクトが格納されているか簡単にチェックできます」

図16　bool関数5…文字列を指定した場合

▶数値を代入した場合

bool関数のオブジェクトに数値を代入することで真偽判定できます．図17のようにオブジェクトに0を代入したときにFalse，0以外の数値(整数や小数)ではTrueを返します．それぞれ，type関数でbool関数が返すオブジェクトの型を確認するとbool型でした．

```
[1]  print(bool(0))
     False

[2]  type(bool(0))
     bool

[3]  print(bool(1))
     True
```

（Trueや Falseの代わりに1や0が使えます）

```
type(bool(1))
bool
```

図17　bool関数6…数値を代入した場合

▶bool型で計算する

bool型はint型のサブクラスなので数値のように計算できます．bool関数の中でbool型のオブジェクトを計算して結果を返す例を**図18**に示します．

```
print(bool(True + False))
True
print(bool(True - True))
False
```

（複数の条件式の結果からTrueやFalseが返されるので，総合的に評価するときに使えます）

図18　bool関数7…bool型で計算する

2-4　文字列を扱うstr関数

● str関数の利用目的

str関数は，文字列以外（主に数値，小数，日付，時刻など異なるオブジェクト型）のオブジェクトを文字列に変換します（**図19**）．文字列の画面表示に使われるprint関数と組み合わせて表示に使用したり，リストや辞書の要素を文字列に変換したりする際によく利用します．

ここでは紹介しきれませんが，str関数には40種類以上のサブ関数があります．例えば文字コードの変換，文字列の書式化操作，大文字/小文字などの判定，文字列の一部を置換，文字列の分割などです．

● str関数の書式

str関数の書式は次になります．

```
str(オブジェクト)
str(オブジェクト, encoding='utf-8',
                  errors='strict')
```

str関数は文字列以外のオブジェクトを引数として渡すとそれを文字列に変換します．

▶引数

• encoding

変換後の文字列の文字コードを指定できます．デフォルトではutf-8ですが，euc_jp，shift_jisなどに変えられます．

• errors

デフォルトでstrictになっており，エンコーディング・エラーとしてUnicodeErrorを送出します．

指定できる値は他にignore，replace，その他のcodecs.register_error()を通して登録された名前などです．

● 文字列を直接記述する方法

Pythonのテキスト・データはstrオブジェクトの文字列として扱われます．文字列は，ユニコード・ポイントのイミュータブル（作成後にその状態を変えることのできない）なシーケンスです．

str関数を使わずに文字列を記述するには，文字列リテラル（0文字以上の連続した文字列を示す定数）を記述する方法があります．

'か"で前後を囲むと文字列として扱われます．例えば，

```
a = "12345"
```

とした場合，"で囲われた中の値が暗黙的に文字列へ変換されます．

'と"を組み合わせる文字列リテラルの記述方法もあります（**リスト1**）．

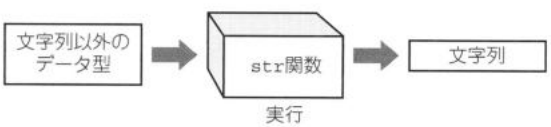


図19　文字列以外のオブジェクトを文字列に変換する

リスト1　複数の方法で文字列を定義できる

```
'"ダブル" クォートを埋め込むことができます'
"'シングル' クォートを埋め込むことができます"
'''3つのシングルクォート'''
"""3つのダブルクォート"""
```

（'の中に"を埋め込む／"の中に'を埋め込む／3重引用符を使って埋め込む）

3重引用符文字列は，複数行に分けることができ，関連付けられる空白は全て文字列リテラルに含まれます．

● 実行例

▶整数

図20はstr関数に整数を代入して文字列に変換し，画面出力する例です．type関数でstr関数が返すオブジェクト型はstr型となっており，文字列です．str型に変換されたオブジェクトは，int型やfloat型のように演算子を使って計算できません．

```
print(str(12345))
12345

[ ]

type(str(12345))
str
```

図20
str関数1…整数

▶小数

str関数のオブジェクトに小数を代入して文字列に変換します．図21のようにtype関数でstr関数が返すオブジェクト型はstr型となっており文字列です．

```
print(str(0.123456))
0.123456

type(str(0.123456))
str
```

図21
str関数2…小数

▶'と"を両方使う

図22のようにstr関数を使って'の中に"を埋め込んで数値を文字列に変換して画面出力できます．逆に"の中に'を埋め込んで数値を文字列に変換して画面出力することもできます．

```
print(str('123"456"6789'))
123"456"6789

print(str("123'456'6789"))
123'456'6789
```

文字列の中に"や'で囲われたテキストを表現することができます．英数字でメッセージを強調表示するときに使えそうです

図22　str関数3…'と"を両方使う

▶3重引用符

str関数で3重引用符を使って埋め込んだ数値を文字列に変換すると，図23のように文字列が連結されて画面出力できます．

```
print(str('''123456'''"""6789"""))
1234566789
```

図23　str関数4…3重引用符

▶日付を読み出す

図24はimport文でdatetimeライブラリを読み込み，datetime.date.today()から返される今日の日付を，str関数で文字列に変換して，print関数で画面出力した例です．

4行目のようにソースコードを記述しても結果は同じです．print関数内で自動的にstr関数を使って文字列に変換しているためです．

6行目のtype関数でdatetime.date.today()が返すオブジェクト型はdatetime.date型ですが，8行目のようにstr関数で変換するとstr型が帰ってきていることが分かります．

オブジェクトがdatetime.date型であれば日付や時刻の計算に使えますが，str型では文字列ですので計算できません．

```
[1]  import datetime
     print(str(datetime.date.today()))
     2021-03-23
4行目
[2]  print(datetime.date.today())
     2021-03-23
6行目
[3]  type(datetime.date.today())
     datetime.date
8行目
     type(str(datetime.date.today()))
     str
```

Pythonは自動的にオブジェクト型を変換してくれるので，型を気にしなくても使えます．
便利でもあり，型を見失いやすくもあるので，型のチェックは大切です

図24　str関数5…日付を読み出す

さとう・せい

佐藤 聖

3-1 オブジェクトの長さや数を取得するlen関数

図1 オブジェクトの長さや要素の数を取得する

● len関数の利用目的

文字列やバイト列，リストや辞書からオブジェクトの長さまたは要素の数を返します(図1)．for文やif文，while文の条件式の中でこれらの数値を比較対象やカウンタなどとして利用します．

● len関数の書式

len関数の書式は次になります．

```
len(オブジェクト)
```

len関数の引数には，シーケンス(文字列，バイト列，タプル，リスト，rangeなど)またはコレクション(辞書，集合，凍結集合など)を代入して，オブジェクトの長さまたは要素の数を返します．

● 実行例

▶文字列の長さ

len関数のオブジェクトに文字列を代入すると文字数が返されます．

図2の例ではそれをprint関数に渡して画面出力しています．英字だけでなく，空白も文字列としてカウントされるので結果として11が表示されます．

```
print(len('Hello World'))
```

```
11
```

文字以外にも空白もカウントされます

図2 長さや数を取得するlen関数1…文字列

▶リストの要素数

リストの要素数をカウントできます．図3の1行目はリストlist1に4つの要素を代入します．2行目でlen関数でlist1の要素数が返され，print関数で画面出力します．リストには4つの要素がありますので，画面には4が表示されます．

```
list1 = ["apple", "orange", "lemon", "banana"]
print(len(list1))
```

```
4
```

for文のループ処理でリストの要素分，処理を繰り返すときに，ループ回数を決めるためによく利用します

図3 長さや数を取得するlen関数2…リスト

▶辞書のオブジェクト数

辞書のオブジェクトをカウントできます．

図4の1行目で変数dict1に4つのキーとオブジェクトとを組み合わせて代入します．

2行目のlen関数で変数dict1をオブジェクトとして代入すると辞書のオブジェクト数が返され，print関数で画面出力します．

辞書のオブジェクトは4つありますので画面には4が出力されます．

```
dict1 = {1:"apple", 2:"orange", 3:"lemon", 4:"banana"}
print(len(dict1))
```

```
4
```

オブジェクトを追加や削除する辞書では，オブジェクト数をプログラムでチェックできます

図4 長さや数を取得するlen関数3…辞書

3-2 リストを扱うlist関数

● list関数の利用目的

リストの作成に利用する関数です．リストは内部要素の値が書き換え可能なオブジェクトであり，ミュータブルなシーケンスと言えます(図5)．一般的に同種の項目の集まりを配列構造として格納するために使われます．

例えば，国語のテストの成績を格納するなどの使い方ができます．

1つのリストに同じ性質のデータ(オブジェクト)としてテストの成績を生徒番号順のように並べる順番のルールを決めて，まとめて格納できます．

また，生徒別に変数を用意してテストの成績を格納するのと比べて，リストを使った方が短いソースコードの記述で済みます．

リストに格納した成績を参照するときにも，生徒番号順に並んでいることが分かっていれば，リストから目的の生徒の成績を参照したり計算に活用したりするのが簡単です．

● list関数の書式

list関数の書式は次になります．

```
list([イテラブルなオブジェクト])
```

図5　同種のデータの集まりからリストを作る

list関数は繰り返し処理に使われるイテラブルなオブジェクトを代入することでリストを返します．

イテラブルなオブジェクトとは，要素を1回に1つずつ返す機能を持ったオブジェクトです．

例を次に示します．

- 全てのシーケンス型オブジェクト(リスト，タプル，文字列)
- 非シーケンス型オブジェクト(辞書，ファイル・オブジェクト)
- __iter__()関数を持つオブジェクト
- __getitem__()関数を持つオブジェクト

● list関数を使わずにリストを取得する方法

list関数を使う以外にも以下の方法でリストを構成できます．

- **角括弧の対を使って空のリストを表す**
  []
- **角括弧を使って項目をカンマで区切る**
  [オブジェクト]，[オブジェクト，オブジェクト，オブジェクト]
- **リスト内包表記を使う**
  [変数 for 変数 in イテラブルなオブジェクト]

● 実行例

▶文字列

図6の1行目でlist関数に文字列のオブジェクトを代入するとリスト型のオブジェクトを返します．

それをprint関数で画面出力するとabcの文字列は，1文字ごとに個別のオブジェクトとしてリストが作られます．

3行目でtype関数でオブジェクトの型を確認するとlist型です．

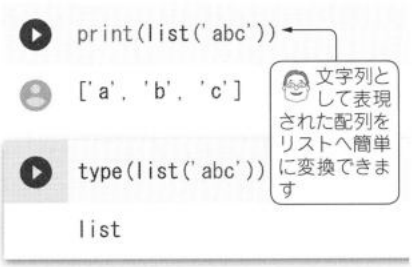


図6
リストを扱うlist関数1…文字列

▶タプル

list関数を使ってタプル型のオブジェクトをリスト型に変換できます．

図7の1行目でtype関数でタプルの型を調べると，2行目のようにtuple型です．

3行目でタプルをlist関数のオブジェクトとして代入するとリストに変換されます．それをprint関数を使って画面出力しています．

5行目でtype関数で確認するとlist型となっており，リスト型に変換されたことが確認できます．

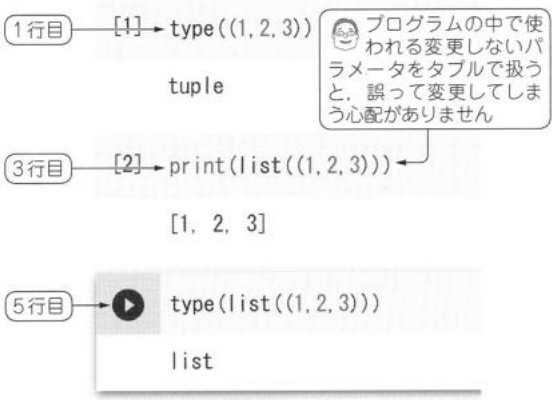


図7　リストを扱うlist関数2…タプル

3-3　ループなどに利用するrange関数

● range関数の利用目的

range関数を使うとlist関数と組み合わせて等差数列のリストを作成できます(**図8**)．一般に，リストを作るよりもrange関数を使ったほうが短いソースコードを記述できます．

for文で特定の回数だけループ処理するときに，range関数で返されるシーケンス(連番)が使われます．

図8　ループ処理などで使う等差数列のオブジェクトを作る

通常のリストやタプルとの違いとして，range関数が返すrange型は，シーケンスのサイズや，表す

範囲にかかわらず常に一定ということがあります．

メモリを少量しか消費しないため，ラズベリー・パイなどの組み込み系コンピュータのように，メモリ容量に限りのあるデバイスで利用するときにより効果を発揮します．

● range関数の書式

range関数の書式は次になります．

```
range(終了)
range(開始，終了)
range(開始，終了，ステップ)
```

range関数の引数には整数で開始，終了，ステップを設定できます．終了を設定すると，数のイミュータブルなシーケンス(内部要素の値が書き換え不可なオブジェクト)を返します．

▶終了だけを設定する場合

range関数の引数は，終了だけを設定して開始とステップを省略できます．その場合，開始(初項)を設定しないとデフォルトの0になり，range(5)のようにすると0～4のシーケンスを返します．

シーケンスは0～4の配列を返すという意味ではなく，range関数が呼び出されるたびに0～4のシーケンスの先頭から順に値を返す処理を行いますので，関数実行のたびに0，1，2，3，4の異なる値を返します．

▶開始と終了の値を設定する場合

range(2，5)のように記述できます．関数は2～4のシーケンスを返すので，関数が実行されるたびに2，3，4を順に返します．

range(0，0)やrange(0)のときは，range(0，0)が返されます．

▶ステップも設定する場合

rangeが返すシーケンスの範囲が開始の0から終了(終了の数－1)までになる理由は，ステップ(公差)の仕様のためです．

ステップにはプラスおよびマイナスの数を指定できます．ステップのデフォルトは1なので，この幅で増減されます．

ステップが0の場合，ValueErrorが送出されます．

ステップが正の場合，range(2，5，1)なら開始＋ステップ数×シーケンス番号が計算されるので，関数が実行されるたびに，

$2+1\times0=2$
$2+1\times1=3$
$2+1\times2=4$

のように処理されます．

ステップが負の場合も，range(5，2，-1)なら開始＋ステップ数×シーケンス順が計算されるので，関数が実行されるたびに，

$5-1\times0=5$
$5-1\times1=4$
$5-1\times2=3$

のように処理されます．

● 実行例

▶終了だけ設定

図9の1行目でrange関数に終了として11を代入した場合，未指定の開始とステップは，開始のデフォルトの0と，ステップのデフォルトの1が代入されます．

type関数でrange関数が返すシーケンスを確認するとrange型です．

3行目でrange関数により数のイミュータブルなシーケンスが返され，list関数を使ってlist型に変換して，print関数で画面出力しています．

画面には0～10の範囲のリストが出力されます．

```
type(range(11))
range
print(list(range(11)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

ループ処理で連番の数字を生成するときによく使います．簡単にループ回数を設定できます

図9　ループなどに利用するrange関数1…終了だけ設定

▶開始，終了，ステップを設定

図10はrange関数で開始，終了，ステップを指定した場合の使い方の例です．開始に1(デフォルトと同じ値)，終了に11，ステップに1(デフォルトと同じ値)を指定します．

rangeの範囲は1から10になるまで1ずつ増える数のシーケンスになります．これをlist関数でrange型からリスト型に変換し，print関数で以下のようにリストを画面出力します．

```
print(list(range(1, 11, 1)))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

図10　ループなどに利用するrange関数2…開始，終了，ステップを設定

▶ステップを変えてみる

図10の例のステップを3に変えると，**図11**のように開始の1から3ずつ増えた数のリストを終了の11未

満の範囲で出力します．

rangeの範囲は等差数列になり，開始の1＋0（ステップの3が0個），開始の1＋3（ステップの3が1個），開始の1＋6（ステップの3が2個），開始の1＋9（ステップの3が3個）のようにシーケンスが作られます．

このシーケンスはlist関数でリスト型に変換され，print関数で画面出力されます．

```
print(list(range(1, 11, 3)))
[1, 4, 7, 10]
```

偶数や奇数の数列も簡単に作れます

図11　ループなどに利用するrange関数3…ステップを変える

▶マイナスのシーケンス

ステップにはマイナスの数を代入すると，開始の数から終了の数までステップで指定した幅で減少していくシーケンスを作れます．**図12**の例のようにrange関数が返すシーケンスをlist関数でリストにすると画面には以下のように表示されます．

```
print(list(range(10, 0, -1)))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
print(list(range(10, 0, -2)))
[10, 8, 6, 4, 2]
```

降順の数列も一行のソースコードで生成できます

図12　ループなどに利用するrange関数4…マイナスのシーケンス

▶終了を0とした場合

range関数の引数として終了を0としたときにr[0]が値の制約を満たさないので，**図13**のようにrange(0, 0)が返されます．シーケンスにオブジェクトが含まれていないため，list関数に渡されると空のリストを出力します．

print関数では空のリストであるカッコだけを画面出力します．

```
print(list(range(0)))
[]
```

図13
ループなどに利用するrange関数5…終了を0とした場合

▶小数のシーケンス

range関数では小数を扱えません．for文を使えば繰り返し計算することで，小数のリストが作れます．なお，for文やappend関数の使い方は後ほど解説します．

例えば**図14**のようにソースコードを書くことができます．変数list1に空のリストを作り，for文で1～5のシーケンスから値を1つずつ取り出します．その値を10で割り，list1の要素として追加し，最後にprint関数でlist1の要素を画面表示します．

```
list1 = []
for x in range(1,6):
    list1.append(x / 10)
print(list1)
[0.1, 0.2, 0.3, 0.4, 0.5]
```

図14
ループなどに利用するrange関数6…小数のシーケンス

▶九九表を作る

for文で2重の繰り返しで九九表を作ることができます．**図15**のように変数numに1～9のシーケンスを代入し，初めのfor文でシーケンスから掛けられる数を取り出し，変数list1を空にします．

2つ目のfor文で掛ける数を取り出して，掛け算の結果をlist1に格納後，2つ目のfor文の処理が終わったらprint関数でlist1のリスト・オブジェクトを画面表示します．この繰り返しで九九表を表示できます．

```
num = range(1,10)
for x in num:
    list1 = []
    for y in num:
        list1.append(x * y)
    print(list1)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 4, 6, 8, 10, 12, 14, 16, 18]
[3, 6, 9, 12, 15, 18, 21, 24, 27]
[4, 8, 12, 16, 20, 24, 28, 32, 36]
[5, 10, 15, 20, 25, 30, 35, 40, 45]
[6, 12, 18, 24, 30, 36, 42, 48, 54]
[7, 14, 21, 28, 35, 42, 49, 56, 63]
[8, 16, 24, 32, 40, 48, 56, 64, 72]
[9, 18, 27, 36, 45, 54, 63, 72, 81]
```

図15　ループなどに利用するrange関数7…九九表を作る

3-4　オブジェクトを並べ替えるsort関数

図16　オブジェクトの要素を並べ替える

● sort関数の利用目的

list関数（listクラス）で提供されるsort関数です．リストや辞書の要素やオブジェクトを並び替えるのに利用します（図16）．シーケンスを並び変えるときの容量節約のため，シーケンスの要素番号を変更できます．また，この関数では並び替えしたシーケンスを返しません．

● sort関数の書式

sort関数の書式は次になります．

```
sort(*, key=None, reverse=False)
```

sort関数は，シーケンスのオブジェクト間を比較演算子<で比較して，リストや辞書の要素やオブジェクトの位置を並び替えます．リストの要素は同じオブジェクト型でないと比較できません．

例外は抑制されず，比較演算がどこかで失敗したら，ソート演算自体が失敗します．リストは部分的に並び変えられた状態で残され，並び替え途中の状態が保持されます．

▶引数

・key

sort関数の引数はキーワードで渡します．keyのデフォルトはNoneで，別のキー値を計算せず，リストの値が直接並び替えられます．また，リストのそれぞれの要素から比較キーを取り出すのに引数keyを使います．例えば，

```
int, float, len, str.lower, lambda式
```

などを指定でき，関数を定義してkeyに関数名を記述することで独自キーを使った並び替えも行えます．

・reverse

真偽値（bool値のTrue，False）を指定することで，リストの要素の並び順を昇順や降順に変えられます．デフォルトの値はFalseで，昇順を意味します．Trueを指定した場合は降順に並び替えます．

● 実行例

▶sort関数のデフォルト動作で並べ替え

sort関数でリストをデフォルトの昇順で並び替える例です．

図17の1行目で変数list1にリストを格納し，2行目でsort関数に引数を指定せずに，デフォルトのkey=None，reverse=Falseとしてリストの要素を並び替えます．この処理によってオブジェクトの要素位置が並び替えられています．

```
list1 = ['apple', 'lemon', 'orange', 'banana']
list1.sort()
print(list1)
```

リストの要素を並び替えて整頓するのによく使います

```
['apple', 'banana', 'lemon', 'orange']
```

図17　オブジェクトを並べ替えるsort関数1…デフォルト動作で並べ替え

▶文字数で並べ替え

sort関数の引数keyには文字列の長さを比較するlen関数を指定できます．

図18の1行目で変数list1にリストを格納し，2行目のsort関数の引数としてlenを渡します．リストの0は1文字，0123は4文字，01は2文字，012は3文字として文字列をカウントされます．文字数の少ない方から多い方へ順に並べ替えられ，print関数が画面出力します．

4行目の結果を見ると，文字数による並び替えができています．

```
list1 = ['0', '0123', '01', '012']
list1.sort(key=len)
print(list1)
```

```
['0', '01', '012', '0123']
```

図18　オブジェクトを並べ替えるsort関数2…文字列で並べ替え

▶整数で並べ替え

ここでは，sort関数でリスト内の数値を昇順に並び替えます．

図19の2行目のように引数のkeyにintを指定できます．変数list1のオブジェクトはsort関数からint関数に渡され，整数に変換された後，sort関数で並べ替えます．実は，list1.sort()としても同じ結果です．

3行目のprint関数で変数list1を画面表示しています．

```
list1 = ['1000', '10', '1', '100']
list1.sort(key=int)
print(list1)
```

```
['1', '10', '100', '1000']
```

図19　オブジェクトを並べ替えるsort関数3…整数で並べ替え

▶小数で並べ替え

引数のkeyには，図20のようにfloat関数を指定できます．図19の例と同じようにlist1.sort()としても同じ結果です．keyに設定する関数が返すオブジェクトを利用してオブジェクトの並び替えが行えます．

```
list1 = ['0.1', '12.3', '0.0']
list1.sort(key=float)
print(list1)
```

```
['0.0', '0.1', '12.3']
```

図20　オブジェクトを並べ替えるsort関数4…小数で並べ替え

▶大文字・小文字を考慮した並べ替え

リストに含まれる文字列を並び替えできます．結果は，大文字，小文字の順になり，図21の4行目の画面出力結果はdanが1番最後です．

```
list1 = ['Ronit', 'Dan', 'dan']
list1.sort()
print(list1)
```

```
['Dan', 'Ronit', 'dan']
```

図21　オブジェクトを並べ替えるsort関数5…大文字・小文字を考慮した並べ替え

▶アルファベット順で並べ替え

デフォルトの文字列の並べ替えの優先順位は，アルファベット順，その次に大文字，最後に小文字の順です．

文字の大文字や小文字には関係なく，アルファベット順で並び替えたいこともあります．sort関数の引数keyにstr.lowerを指定すると大文字は，小文字として扱われ並び替えできます．

結果，図22の4行目のように画面出力されます．なお，str.lowerの代わりにstr.upperを指定すると小文字は大文字として扱われます．

```
list1 = ['Ronit', 'Dan', 'dan']
list1.sort(key=str.lower)
print(list1)
```

```
['Dan', 'dan', 'Ronit']
```

SNSからユーザ名を取得したり，ECサイトから商品名を取得したりしたときなど，読み方で名寄せするときに使えます

図22　オブジェクトを並べ替えるsort関数6…アルファベット順で並べ替え

▶キーに指定したstr.lower関数を見てみる

sort関数では他クラスの関数を呼び出して利用するときに，keyに指定して利用できます．ですので，str.lowerは関数として利用できます．図23のように関数にABCを代入するとabcが返され，print関数で画面出力します．

図23　オブジェクトを並べ替える…str.lower関数を見てみる

```
print(str.lower('ABC'))
```

```
abc
```

▶str.upper関数を見てみる

str.lowerとは逆に文字列を大文字に変換します．

図24の1行目でstr.upperの引数として与えられたabcが，ABCに変換され，print関数で画面出力されています．

図24　オブジェクトを並べ替える…str.upper関数を見てみる

```
print(str.upper('abc'))
```

```
ABC
```

▶降順で並べ替え

降順に並び替えしたい場合もあります．その場合，図25の2行目のようにsort関数の引数reverseにTrueを指定します．デフォルトはFalseで昇順ですが，Trueは降順に並び替えます．

4行目のようにリストの要素は大きい数から小さい数へ並び変えられます．

```
list1 = ['1000', '10', '1', '100']
list1.sort(reverse=True)
print(list1)
```

```
['1000', '100', '10', '1']
```

引数のkeyとreverseの組み合わせで多様な並び替えができます．keyには，自作した関数を使用することもでき，独自の並び替えも行えます

図25　オブジェクトを並べ替えるsort関数7…降順で並べ替え

3-5　要素を並び替えるsorted関数

図26　元のオブジェクトは変更せずに並べ替えたオブジェクトを返す

● sorted関数の利用目的

sorted関数は，リストやタプルの要素の相対順序を変更せずに結果を返します．sort関数のように並び替えによる要素位置を変更しません(図26)．タプルの要素を並び替え，複数パスからリストの要素順序を変えずにアクセスする際に利用されます．

● sorted関数の書式

sorted関数の書式は次になります．

```
sorted(オブジェクト, *, key=None, reverse=False)
```

sorted関数の引数に，オブジェクト，key，reverseを指定できます．*には指定できる変数はなく省略できます．この関数ではオブジェクトやイミュータブルなシーケンス(内部要素の値が書き換え不可なオブジェクト)からイテラブルなシーケンスを作り(元シーケンスのコピーのようなもの)，要素を並び替えて返します．引数のkeyとreverseはsort関数と同じように扱うことができます．

● 実行例

▶タプルのソート

図27の1行目でタプルの要素に文字列を指定します．

2行目でsorted関数にタプル型のシーケンスが格納されている変数tuple1を代入して結果をprint関数で画面出力します．結果は括弧が()ではなく[]に変わります．

4行目でtype関数でsorted関数が返すオブジェクト型を確認するとlist型です．

```
tuple1 = ('abbb', 'abcc', 'aaaa', 'abcd')
print(sorted(tuple1))
```

```
['aaaa', 'abbb', 'abcc', 'abcd']
```

```
type(sorted(tuple1))
```

```
list
```

図27　要素を並び替えるsorted関数1…タプルのソート

▶元のリストは変更されない

図28は1行目で数字のリストを作成して変数list1に代入して，2行目でsorted関数の引数reverseをTrueにして降順に並び替えてprint関数で画面出力します．想定通り，数字が並び変えられています．

3行目の結果はイテラブルなシーケンスの画面出力になります．

4行目でprint関数で元の変数list1を代入すると，5行目の画面出力では1行目のリストと同じ順番でオブジェクトが並んでいます．つまり元のオブジェクトの順序が保持されています．

2行目のsorted関数では1行目のリストからシーケンス型オブジェクトをコピーしています．3行目で出力されている結果は保持されていませんので，再利用するときにはsorted関数が返すオブジェクトを変数に代入しておく必要があります．

```
list1 = ['10', '100', '1', '0']
print(sorted(list1, reverse=True))
```

1行目

```
['100', '10', '1', '0']
```

元の並び順が保持されるので，データの並び順に意味を持たせていて，オブジェクトの追加や削除をしたいときに便利です

```
print(list1)
```

5行目

```
['10', '100', '1', '0']
```

図28　要素を並べ替えるsorted関数2…元のリストは変更されない

▶並び替えた結果を保持するには変数に代入しておく

図29の1行目のように，変数list1に格納された先ほどのリストをsorted関数で並び替えます．その結果を変数list2に格納します．そうすると変数list2には，並び替え結果が保持されます．

2行目でprint関数を使って変数list2を画面出力しています．それを見ると，並べ替えた後のリストになっています．

```
list2 = sorted(list1, reverse=True)
print(list2)
```

```
['100', '10', '1', '0']
```

図29　要素を並べ替えるsorted関数3…結果を保持するには変数に代入する

さとう・せい

ダウンロード・データあります

佐藤 聖

4-1 Pythonの式

リスト1 Pythonで使える演算子

```
+  -  *  **  /  //  %  @  <<  >>  &  |  ^  ~  <  >  <=  >=  ==  !=
```

リスト2 Pythonで区切り文字として使われる記号

```
(  )  [  ]  {  }  ,  :  .  ;  @  =  ->  +=  -=
*=  /=  //=  %=  @=  &=  |=  ^=  >>=  <<=  **=
```

Python言語の式について知らなければならないことは，次のように多くあります．

算術変換，アトム，原子的要素，プライマリ，Await式，べき乗演算，単項算術演算とビット単位演算，二項算術演算，シフト演算，ビット単位演算の二項演算，比較，ブール演算，条件式，ラムダ，式のリスト，評価順序，演算子の優先順位

などです．しかしプログラミング入門者が，これらをすぐに利用することは少ないと思います．Pythonに慣れてくるとライブラリやパッケージを利用したソースコードを記述することが多くなると思いますので，最初からこれら全てを覚える必要はありません．

本記事では，よく利用される式だけを取り上げます．また，使い方を紹介する式には利用方法以外にもソースコードの記述方法があります．将来的に必要になったところで，そうした利用法を学ぶとよいと思います．

Pythonで使われる記号

他のコンピュータ言語と同じようにPythonでも1つまたは複数の記号を組み合わせて作られる，プログラム上の意味を持つトークンと呼ばれるキーワードのようなものを使います．トークンにはいろいろな記号が使われます．

● 演算子

公式ドキュメント(1)で紹介されている演算子をリスト1に示します．利用方法は後ほど紹介します．

● 区切り文字としての機能を持つデリミタ

リスト2に，文法上のデリミタ(区切り文字)として働くトークンを示します．

ピリオド(.)は浮動小数や虚数リテラル中にも置けます．

リスト中の累算代入演算子は字句的には，デリミタとしての振る舞いだけでなく，演算子としても働きます．詳しくは後ほど紹介します．

● 特殊な意味を持つ記号

以下の記号は印字可能なASCII文字で，他の記号と組み合わせることで特殊な意味を持つトークンとなったり，字句解析器にとって重要な意味を持ったりします．

```
'  "  #  \
```

▶シングル・クォートとダブル・クォート

シングル・クォート(')とダブル・クォート(")は文字列を作るときに使用します

▶コメント文

シャープ(#)は，コメント文をソースコードに記述するときに利用します．

▶エスケープ・シーケンス

バックスラッシュ(\)はエスケープ・シーケンスと

表1 演算子の優先順位

演算子	説　明
lambda	ラムダ式
if -- else	条件式
or	ブール演算OR
and	ブール演算AND
not x	ブール演算NOT
in, not in, is, is not, <, <=, >, >=, !=, ==	所属や同一性のテストを含む比較
|	ビット単位OR
^	ビット単位XOR
&	ビット単位AND
<<, >>	シフト演算
+, -	加算および減算
*, @, /, //, %	乗算，行列乗算，除算，切り捨て除算，剰余
+x, -x, ~x	正数，負数，ビット単位NOT
**	べき乗
await x	Await式
x[index], x[index:index], x(arguments...), x.attribute	添字指定，スライス操作，呼び出し，属性参照
(expressions...), [expressions...], {key: value...}, {expressions...}	結合式または括弧式，リスト表示，辞書表示，集合表示

してアルファベットや数字と組み合わせて利用します．

● 使えない記号

以下の印字可能なASCII文字は，Pythonでは使われていません．これらの文字が文字列リテラルやコメント以外の場所にあるとエラーになります．

```
$　?　`
```

演算子の優先順位

演算子には優先順位があり，式を正しく組み立てるときに考慮する必要があります．演算子の優先順位の要約は**表1**のようになります(2)．表の上側ほど優先順位が低く，下にあるものほど高くなります．

また，同じ行にある演算子は同じ優先順位です．計算の順序がある場合，括弧を使って優先順位に高低を付けます．

比較，所属，同一性のテストで使われる演算子は全て同じ優先順位で，左から右に連鎖するという特徴があります．

4-2　算術演算子

表2　Pythonで使える算術演算子

書　式	説　明
+a	正数
-a	負数
a + b	足し算
a - b	引き算
a * b	掛け算
a / b	割り算
a % b	aをbの余り
a ** b	aのb乗
a // b	aとbの切り捨て除算
a = b	bをaに代入

表3　Pythonで使える代入演算子

書　式	短縮形	説　明
a = a + b	a += b	aとbの足し算の結果をaに代入
a = a - b	a -= b	aとbの引き算の結果をaに代入
a = a * b	a *= b	aとbの掛け算の結果をaに代入
a = a / b	a /= b	aとbの割り算の結果をaに代入
a = a % b	a %= b	aとbの剰余算の結果をaに代入
a = a ** b	a **= b	aのb乗の結果をaに代入
a = a // b	a //= b	aとbの切り捨て除算の結果をaに代入

● 演算子とその説明

Pythonは，式を左から右へと順に評価します．代入式を評価するときは，右辺が左辺よりも先に評価されます．この評価順序を理解していれば，代入演算子もうまく利用できます．

算術演算子は代数演算子とも呼ばれ，基本的な数値の計算に利用されます．書式，書式の短縮形，説明は**表2**，**表3**の通りです．学校で習う表記とは異なる部分も多くあります(**表2**)．

計算結果を変数に代入する場合，短縮した記述も可能です．プログラミングでは可読性が良い短縮形が頻繁に利用されます(**表3**)．いずれも結果は変数aに代入されるため，少ないメモリで処理できます．

● 数値の演算実行例

▶正負の整数

図1は正および負の数をprint関数で画面出力した例です．正の数は+を付けなくても正の数として扱われますが，明示的に表現してもエラーにはなりません．

```
print(+1)
1
print(-1)
-1
```

オブジェクト型を指定しなくても，数字として扱われます

図1　算術演算子1…正負の整数

▶足し算および引き算

図2は足し算/引き算の結果をprint関数で画面出力した例です．print関数だけでなく，他の関数へも式をオブジェクトとして渡せます．同じ型のオブジェクトが代入された変数同士なら式を組み立てて計算できます．

```
print(10 + 1)
11
print(10 - 1)
9
```

数字と演算子の間に半角スペースがなくても計算されます．
半角スペースが入ると読みやすいです

図2　算術演算子2…足し算および引き算

▶掛け算

掛け算は**図3**のように計算されます．掛け算で使用する×（掛け算記号）は，Pythonでは*（アスタリスク）で記述します．

```
print(10 * 2)
```
```
20
```

スペースは半角で！全角スペースではエラーになり，プログラムが止まってしまいます

図3　算術演算子3…掛け算

▶割り算

割り算で使用する÷（除算記号）は，Pythonでは/（スラッシュ）で記述します．割り算を実行すると5ではなく，**図4**のように5.0と結果が返ってきます．Python 3では割り算の結果が小数で返ってくるのは仕様です．

```
print(10 / 2)
```
```
5.0
```

図4　算術演算子4…割り算

▶剰余算

11を2で割った余りの計算は**図5**のように計算し，整数が返されます．

```
print(11 % 2)
```
```
1
```

図5　算術演算子5…剰余算

▶べき乗

例えば2の3乗であれば，**図6**のように計算されます．

```
print(2 ** 3)
```
```
8
```

図6　算術演算子6…べき乗

▶切り捨て除算

図7のように割り算の結果の整数部を返します．

```
print(7 // 2)
```
```
3
```

図7　算術演算子7…切り捨て除算

● オブジェクト同士の演算実行例

▶代入

図8は変数bのオブジェクトを変数aに代入する例です．変数aにはオブジェクトだけでなく，オブジェクト型も代入されます．

```
b = 2
a = b
print(a)
```
```
2
```

変数はオブジェクトを格納する入れ物として使えます

図8　算術演算子8…オブジェクトの代入

▶足し算

同じオブジェクト型であれば変数間の計算が可能です．**図9**は変数aと変数bのオブジェクトで足し算した結果を変数aに代入する例です．

```
a = 5
b = 2
a += b
print(a)
```
```
7
```

図9　算術演算子9…オブジェクト同士の足し算

▶引き算

図10は変数間の引き算の例になります．計算結果は変数aに代入されます．

```
a = 5
b = 2
a -= b
print(a)
```
```
3
```

図10　算術演算子10…オブジェクト同士の引き算

▶乗除算

図11は変数aと変数bの掛け算および割り算の結果を変数aに代入する実行例です．掛け算では10，割り算では2.5が返ってきます．

```
a = 5
b = 2
a *= b
print(a)
```
```
10
```

変数へのオブジェクトの代入と計算式を分けるとソースコードの変更が簡単になります

```
a = 5
b = 2
a /= b
print(a)
```
```
2.5
```

図11　算術演算子11…オブジェクト同士の乗除算

▶剰余算

図12は変数aを変数bで割った余りを求めて結果を変数aに代入しています.

```
a = 5
b = 2
a %= b
print(a)
```

```
1
```

図12
算術演算子12…
オブジェクト同士
の剰余算

▶べき乗

図13はべき乗の実行例です. 変数bの値が乗数として扱われます.

```
a = 5
b = 2
a **= b
print(a)
```

```
25
```

図13
算術演算子13…
オブジェクト同士
のべき乗

▶切り捨て除算

図14は変数aと変数bで切り捨て除算した結果を変数aに代入する実行例です.

```
a = 5
b = 2
a //= b
print(a)
```

ここでは, 簡単な式で記述していますが, 算術演算子を多数使った複雑な式を組み立てて計算することもできます

```
2
```

図14
算術演算子14…
オブジェクト同士
の切り捨て除算

4-3　比較演算子

● 比較演算子の利用目的

比較演算子は, 数値や文字列の比較に利用されます. 結果はブール型(TrueまたはFalse)で返されます. 主にif文, for文, while文などの条件式で真偽の判定に利用されます.

● 書式と説明

比較演算子を表4に示します. !=, <=, >=などは1対1の比較に利用されます.

以下の演算子は2つのオブジェクトの値を比較しますが, それらは同じオブジェクト型である必要はありません.

```
<, >, ==, >=, <=, !=
```

▶inおよびnot in

この演算子はオブジェクトの所属に対するテストを行います. x in sの評価は, xに数字や文字列, sにリスト, タプル, 集合, 辞書などを指定できます. xがsの要素であればTrueとなり, そうでなければFalseとなります.

x not in sは, x in sの否定を返します.

▶isおよびis not

この演算子はオブジェクトの同一性に対するテストを行います. x is yは, xとyが同じオブジェクトのときにTrueになります. x is not yは, x is yが返すブール値を反転したものになり, xとyが同じオブジェクトのときにFalseになります.

表4　2つのオブジェクトを比べる比較演算子

書　式	説　明
a == b	変数aと変数bが等しい
a != b	変数aと変数bが等しくない
a > b	変数aは変数bより大きい(変数bは変数a未満)
a >= b	変数aは変数bより大きいか等しい(変数bは変数a以下)
a < b	変数aは変数bより小さい(変数bは変数aを超過)
a <= b	変数aは変数bより小さいか等しい(変数bは変数a以上)

● 実行例

▶2つの要素の比較

図15は数値と文字列の等しいときの実行例です.

1～2行目の演算子==は, 両辺が等しいときにTrueを返します. 両辺は同じ数字ですのでTrueが返され, print関数で画面出力されます.

3～4行目では文字列同士の比較になり, 両辺は同じ文字列なのでTrueが返され, print関数で画面出力されます.

5～6行目は数字と文字列の比較ですが, 両辺が同一オブジェクトではないためFalseが返されます.

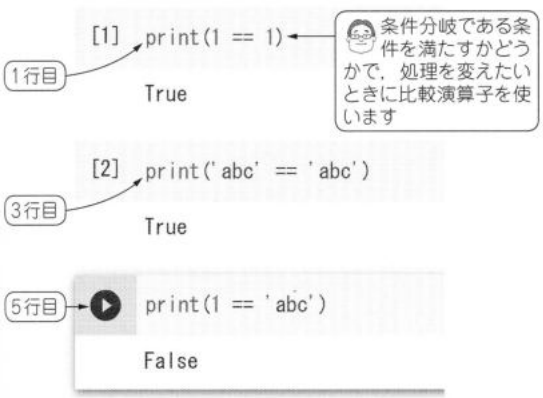


図15　比較演算子1…2つの要素の比較

▶リスト同士の比較

図16ではリスト自体がオブジェクトとして比較されています．両辺の要素が同じであればTrueが返されます．

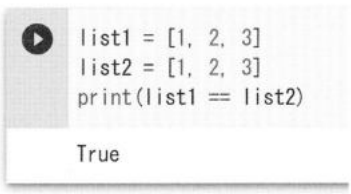


図16
比較演算子2…リスト同士の比較（両辺の要素が同じ）

例えば，図17のように変数list1と変数list2のオブジェクトが異なるとFalseが返されます．

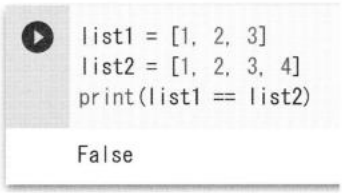


図17
比較演算子3…リスト同士の比較（両辺の要素が異なる）

演算子!=は両辺が等しくないときにTrueを返し，等しいときにFalseを返します（図18）．

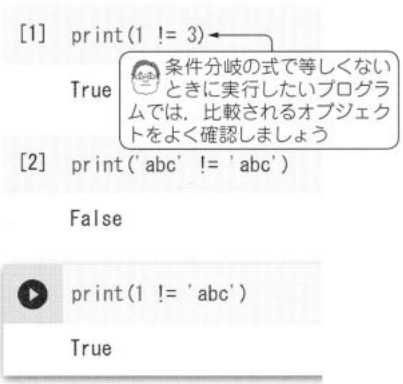


図18
比較演算子4…リスト同士の比較（演算子!=）

図19は数字の比較になり，それぞれ3は5未満，3は5以下，3は5を超過，3は5以上の評価を行っています．小学校で習う不等号と同じです．

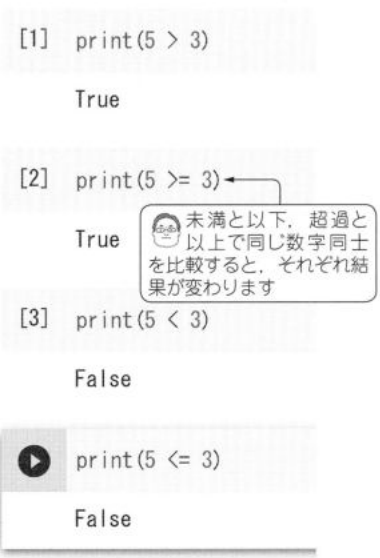


図19
比較演算子5…リスト同士の比較（不等号）

同じ演算子でも数字と文字列では評価方法が異なります．上記の未満，以下，超過，以上は，文字列では等価の文字列が含まれるかどうかを評価します．同じ演算子でも評価するオブジェクトによって返される結果が異なります（図20）．

17～20行目のようにCとcの比較で，比較演算子が>のときにFalse，比較演算子が<のときにTrueが返されます．同じ文字でも大文字と小文字で大小関係が異なります．

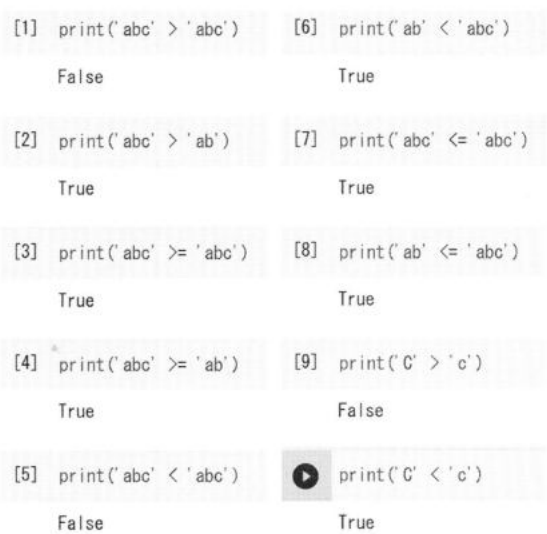


図20　比較演算子6…同じ演算子でも評価するオブジェクトによって演算結果が変わる
文字列同士の比較では，独特なロジックで結果が返される．プログラムが想定外の挙動にならないようしっかりテストしておく

▶所属関係を評価

x in sの評価は，右辺のオブジェクト(s)に左辺のオブジェクト(x)が含まれているとTrueです．

図21の1～3行目では変数aに代入されているabcdeにdが含まれているので評価としてTrueが返されます．

また，4～6行目では変数aにzが含まれていないので評価はFalseを返します．

7～9行目ではリストの要素にbcが含まれているのでTrueを返します．

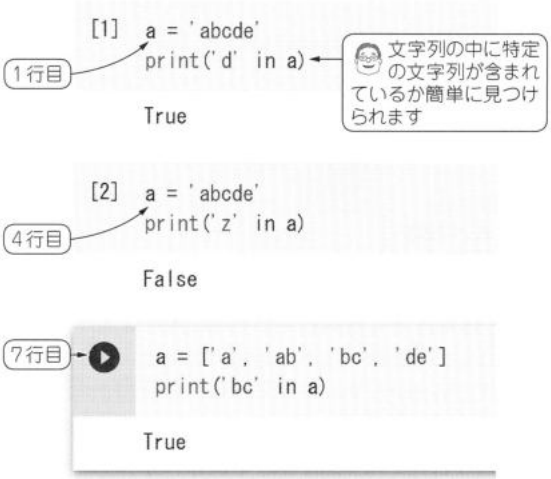


図21　比較演算子7…所属関係を評価1(演算子x in s)

演算子not inの評価結果は，演算子inの反対のブール値を返します．図22のようにabcdeにdが含まれるのでFalseが返されます．

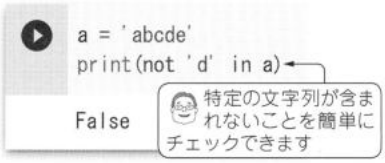


図22
比較演算子8…所属関係を評価2
(演算子not in)

演算子isは両辺のオブジェクトが同一のときにTrueを返します．

図23の1～3行目では変数aに格納された文字列とabcdeが同一であるためTrueです．

4～5行目は，演算子is notを使っており，演算子isの反対の結果を返します．

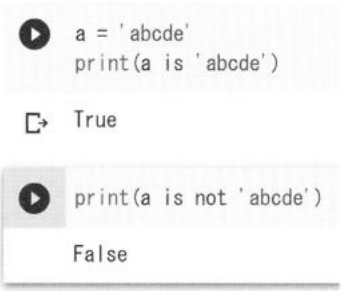


図23
比較演算子9…所属関係を評価3
(演算子is)

4-4　ブール演算子

● ブール演算子の利用目的

ブール演算子は，if文，for文，while文で条件式の中で真偽を判定するのによく用いられます．

● 書式と説明

ブール演算子は，True，False，数値または文字列の真偽を評価できます(表5)．True，Falseもしくは文字列を使った演算は式の前方から処理されます．左辺のオブジェクトを評価したときにTrueかFalseかによって，演算結果が変わります．ブール演算子の振る舞いは，オブジェクトの種類でも異なります．

表5　真偽判定に使われるブール演算子

書　式	説　明
a and b	aがFalseならa，そうでなければb
a or b	aがTrueならa，そうでなければb
not a	aがFalseならTrue，そうでなければFalse

● 実行例

▶数字のand

ブール演算子andは，両辺が数字のときに0はFalse，それ以外の数字はTrueとして評価されます．

図24の1～4行目は左辺のオブジェクトの評価がTrueなので右辺のオブジェクトを返します．

5～8行目は左辺のオブジェクトの評価がFalseなので左辺のオブジェクトを返します．

▶ブール型のand

ブール型(TrueまたはFalse)を評価した例です．

図25の1～4行目は左辺のオブジェクトがTrueなので右辺のオブジェクトを返します．

5～8行目は左辺のオブジェクトの評価がFalseなので左辺のオブジェクトを返します．

```
[1]  print(1 and 0)
     0
[2]  print(1 and 1)
     1
[3]  print(0 and 0)
     0
     print(0 and 1)
     0
```

図24　ブール演算子1…数値型同士の論理演算and

```
[5]  print(True and True)
     True
[6]  print(True and False)
     False
[7]  print(False and True)
     False
     print(False and False)
     False
```

図25　ブール演算子2…ブール型同士の論理演算and

▶文字列のand

文字列の評価は，空（''）はFalse，1文字以上の文字があればTrueです．

図26の1行目は左辺のオブジェクトが空なので評価はFalseになり，左辺のオブジェクトを返します．

3行目は左辺のオブジェクトの評価がTrueですので右辺のオブジェクトを返します．どちらも結果は空が返されます．

```
print('' and 'abc')
```
1行目 ← 空が返されるので何も表示されない

```
print('abc' and '')
```
3行目 ← 空が返されるので何も表示されない

図26
ブール演算子3…文字列のand

▶数値型のor

ブール演算子orでも左から右へ評価されます．数字の0の評価はFalse，それ以外がTrueです．

図27の1～4行目は左辺のオブジェクトの評価がTrueなので左辺のオブジェクトが返されます．

5～8行目は左辺のオブジェクトの評価がFalseなので右辺のオブジェクトを返します．

▶ブール型のor

図28の1～4行目は左辺のオブジェクトの評価がTrueなので左辺のオブジェクトを，5～8行目は左辺のオブジェクトの評価がFalseなので右辺のオブジェクトを返します．

```
[9]  print(1 or 0)
     1
[10] print(1 or 1)
     1
[11] print(0 or 0)
     0
     print(0 or 1)
     1
```

図27　ブール演算子4…数値型同士の論理演算or

```
[13] print(True or True)
     True
[14] print(True or False)
     True
[15] print(False or True)
     True
     print(False or False)
     False
```

図28　ブール演算子5…ブール型同士の論理演算or

▶文字列のor

文字列が空（''）のときの評価はFalse，1文字以上の文字があれば評価はTrueです．

図29の1行目は左辺のオブジェクトの評価がFalseなので右辺のオブジェクトを，3行目の左辺のオブジェクトの評価がTrueなので左辺のオブジェクトを返します．

```
print('' or 'abc')   ← 1行目
abc
print('abc' or '')   ← 3行目
abc
```

図29
ブール演算子6…文字列のor

▶not

ブール演算子notは，ブール型オブジェクトの評価を反転して返します（図30）．

```
[17] print(not 0)
     True
[18] print(not 1)
     False
[19] print(not 'abc')
     False
     print(not '')
     True
```

図30　ブール演算子7…各型に対する論理演算notの結果

◆参考文献◆

(1) Python公式サイトのドキュメント「Python言語リファレンス」．
https://docs.python.org/ja/3.7/reference/lexical_analysis.html#operators

さとう・せい

ダウンロード・データあります

佐藤 聖

5-1 文字列の操作

● 文字列の利用目的

文字列の作成，文字操作，メッセージなどの画面出力などで利用します．str関数は文字列以外のオブジェクトを文字列に変換できましたが，ここでは直接的に文字列を作る方法を解説します．

● 文字列操作の書式

```
'文字列'
"文字列"
'"文字列1"文字列2'
"'文字列1'文字列2"
'''文字列1''', """文字列2"""
```

Pythonのテキスト・データは，strオブジェクト（文字列）として扱われます．'，"，または3重引用符で前後を囲むことで自動的にstr型になります．

演算子，デリミタ，印字可能なASCII文字などが含まれていても文字列として扱えます．複数の文字列は演算子＋で結合できます．

文字列の中に'や"を含めたいときに同時に両方を含めることはできませんが，'の中に"を，または"の中に'を含めることができます．

3重引用符で囲われた複数の文字列はタプルに格納されます．

● 実行例

▶文字列の結合

演算子＋を使うと文字列同士を結合できます．

図1の1～2行目で変数str1とstr2に文字列を格納し，3行目でそれらを結合して変数str3に格納します．

4行目のprint関数で変数str3に格納されたオブジェクトを画面出力します．

```
str1 = 'Hello World'
str2 = '!'
str3 = str1 + str2
print(str3)
```

```
Hello World!
```

図1 文字列の操作1…文字列の結合

▶文字列と数字の結合

文字列と数字を結合するには，数字を文字列に変換して結合します．

図2の3行目で変数num1に格納された数字をstr関数で文字列に変換した後，演算子＋で文字列同士を結合しています．

4行目のprint関数で変数str1に格納されたオブジェクトを画面出力します．数字は文字列に変換して，オブジェクト型を文字列に合わせることで，他の文字列と結合できます．

```
str1 = 'Hello World'
num1 = 2021
str1 = str1 + str(num1)
print(str1)
```

```
Hello World2021
```

図2 文字列の操作2…文字列と数字の結合

▶3重引用符の使い方

図3の1行目で，3重引用符で囲った2つの文字列を，区切り文字で分けて作ったタプルを変数aに代入しています．

3重引用符として，'または"で文字列の前後を囲い，複数の文字列をカンマで区切ります．文字列が1つだけだと，そのオブジェクトはタプルではなく文字列として変数に代入されます．ただし，区切り文字としては必ずカンマを使用する必要があります．

2行目でtype関数を使って変数aのオブジェクトの型を確かめています．

4行目のprint関数で変数aのオブジェクトを画面出力します．

```
a = '''文字列''', """文字列"""
type(a)
```

```
tuple
```

```
print(a)
```

```
('文字列', '文字列')
```

図3 文字列の操作3…3重引用符の使い方

5-2　リスト…複数のデータをまとめて管理できる要素群

表1　多重のリストも作成できる

書　式	説　明
[]	空のリスト
[オブジェクト1，オブジェクト2，オブジェクト3]	リスト
[オブジェクト1，[オブジェクト2，オブジェクト3]]	2重にネストされたリスト

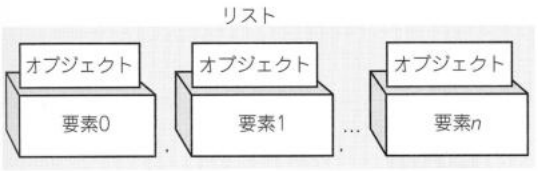


図4　リストとは複数のオブジェクトを並べたデータ構造

● リストの利用目的

リストはシーケンス型の1つです（**図4**）．シーケンス型は他に，タプルやrangeオブジェクトがあります．変数に格納すれば任意の要素を容易に取り出せます．ほとんどのミュータブル（変更が可能な変数の型），イミュータブル（変更が不可能な変数の型）なシーケンスで演算が行え，if文やfor文，while文での条件式で評価に使用できます．

● リストの書式

表1に書式を示します．

リストを作るには幾つかの方法がありますが，ここでは角括弧 [] を使います．数字はオブジェクトとしてそのまま，文字列は'や"で前後を囲んで，リストのオブジェクトに設定できます．リストを構成するときは複数のオブジェクトを格納できます．

リストにオブジェクトを格納せずに空のリストも作れます．空のリストを作ることでリストを初期化し，その後の繰り返し処理の中でリストにオブジェクトを追加する使い方があります．

リストの中に別のリストを格納してネスト構造（入れ子構造）を作れます．リストの中に複数のリストを格納でき，2重のネスト構造だけでなく，3重以上のネスト構造も問題なく作れます．しかし，複雑なネスト構造になると，内部にどんなオブジェクトが格納されているか把握が困難になります．

● 実行例

▶リストの作り方

リストは[と]で囲まれたオブジェクトの列です．**図5**の例では，1，2，3を[1，2，3]のようにしてリストを作ります．

このリストを変数aに代入してprint関数で画面出力すると，画面にはリストがそのまま表示されます．

type関数でこの変数aのオブジェクトの型を見るとlist型になっています．

```
a = [1,2,3]
print(a)
```

```
[1, 2, 3]
```

```
type(a)
```

```
list
```

図5
リストの実行例1…
リストの作り方

▶リストの中から一部を取得

リスト内の一部のオブジェクトだけを表示するときには，リストの要素をスライスします．

図6のように0から始まる要素番号（インデックス番号）を[と]で囲んで指定します．この番号はリスト全体の先頭からの位置です．

```
[1]  print([1,2,3][0])
```

```
1
```

```
[2]  print([1,2,3][1])
```

```
2
```

```
print([1,2,3][2])
```

```
3
```

図6
リストの実行例2…
リストの中から一部を取得

▶リストが格納されている変数をスライス

図7の変数aにはリストが格納されているので，変数名の後ろに[要素番号]のように指定してスライスします．その要素番号に格納されているオブジェクトを参照できます．

```
[1]  a = [1, 2, 3]
     print(a[0])

     1

[2]  print(a[1])

     2

     print(a[2])

     3
```

図7
リストの実行例3…
リストが格納されている変数をスライス

▶リストの一部を範囲指定してスライス

図8の変数aに上記のリストが代入されている状態で，

[開始の要素番号：終了の要素番号]

とすると，リストの探索範囲が設定され，その範囲のオブジェクトを参照できます．

以下の例では，[0：2]を指定した場合，要素番号0と1に格納されたオブジェクト[1，2]が返され，また[0：3]および[0：7]を指定すると[1，2，3]が返されます．それぞれprint関数で画面に出力します．

```
[4]  print(a[0:2])

     [1, 2]

[5]  print(a[0:3])

     [1, 2, 3]

     print(a[0:7])

     [1, 2, 3]
```

図8
リストの実行例4…
リストの一部を範囲指定してスライス

▶リストの前半または後半だけをスライス

リストのスライスは，開始の要素番号だけ，または終了の要素番号だけを指定できます．

探索範囲は開始の要素番号以降または，終了の要素番号以前に設定されます．

また要素番号を指定せずに[：]とすると，探索範囲はリスト全体です．

print関数で画面出力すると**図9**のような結果になります．

```
[3]  print(a[1:])

     [2, 3]

[4]  print(a[:2])

     [1, 2]

     print(a[:])

     [1, 2, 3]
```

図9
リストの実行例5…
リストの前半または後半だけをスライス

▶リストのスライス方法のまとめ

上記のまとめになりますが，リストを作りそこから部分的な要素を抽出する例です．

図10の1行目で1～7の数字を各要素としてリストを作成し，変数list1に代入しています．

2行目のprint関数でlist1のオブジェクトを画面出力します．4行目ではリストの1番目の要素を選択し，print関数で画面出力しています．リストの要素は先頭の0番目～n番目の位置を取り，1番目の要素には数字のオブジェクトとして2が格納されます．そのため，画面出力は2となっています．

6行目ではリストから特定の要素の範囲を指定して参照します．要素番号が1番目から3番目が抽出され，print関数で画面出力します．コロンの前が開始の要素番号，後が終了の要素番号ですが選択範囲は要素番号の4番目ではなく，終了の要素番号－1番目となり，結果として1～3番目の要素位置までが選択範囲です．

リストのオブジェクトから2，3，4が抽出されます．

```
[1]  list1 = [1, 2, 3, 4, 5, 6, 7]
     print(list1)

     [1, 2, 3, 4, 5, 6, 7]

4行目
[2]  print(list1[1])

     2

     print(list1[1:4])

     [2, 3, 4]
```

図10
リストの実行例6…
リストのスライス方法のまとめ

▶リスト同士の演算

リスト同士を演算子＋で結合できます．

図11の1～2行目で変数list1，list2にそれぞれのリストを格納し，3行目で結合して変数

list3に代入します．リストの結合は文字列の結合のように要素の位置を考慮する必要があります．結合は演算子＋の左辺が前方に，右辺が後方に接続します．

4行目でprint関数により変数list3に格納されているリストを画面出力します．結合されたリストが出力されています．

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = list1 + list2
print(list3)
```

```
[1, 2, 3, 4, 5, 6]
```

図11
リストの実行例7…
リスト同士の演算

▶リストの結合

リストは，オブジェクトの型が一致していなくても格納できるのでリスト同士を結合できます．

図12の1～2行目で変数list1には文字列でリストを作成し，変数list2には数字でリストを作成します．

3行目でprint関数の引数の中で変数list1と変数list2を結合して，結果を画面出力します．オブジェクト型が異なっていてもリストを作成できています．

```
list1 = ['apple', 'orange', 'lemon']
list2 = [1, 2, 3]
print(list1 + list2)
```

```
['apple', 'orange', 'lemon', 1, 2, 3]
```

図12　リストの実行例8…リストの結合

▶多重になったリスト

上記ではリストに文字列を追加しましたが，リストを多重にした(ネストした)リストも作れます．多数の変数やリストでオブジェクトを持つよりも，論理的なまとまりとしてリストにまとめる方が扱いが簡単です．

図13の1～2行目で変数list1およびlist2にそれぞれのリストを代入し，3行目で変数list1のリストにappend関数を使って変数list2のリストを追加します．4行目で変数list1のリストをprint関数で画面出力しています．もともとの変数list1の4つ目の要素として，変数list2のオブジェクトが追加されています．

```
list1 = ['apple', 'orange', 'lemon']
list2 = ['strawberry', 'kiwi']
list1.append(list2)
print(list1)
```

```
['apple', 'orange', 'lemon', ['strawberry', 'kiwi']]
```

図13　リストの実行例9…多重になったリスト

▶リストへオブジェクトを追加

図14はリストの最後に新たなオブジェクトを追加する際の操作例です．

1行目でリストを変数list1に代入し，2行目で変数text1に文字列bananaを代入します．

3行目でappend関数を使ってリストの最後にオブジェクトを加えます．書式は，次の通りです．

```
オブジェクトの追加先のリスト.append(オブジェクト)
```

4行目でlist1に格納されているオブジェクトをprint関数で画面出力します．リストの最後にbananaが追加されたことが確認できます．

```
list1 = ['apple', 'orange', 'lemon']
text1 = 'banana'
list1.append(text1)
print(list1)
```

```
['apple', 'orange', 'lemon', 'banana']
```

図14　リストの実行例10…リストへオブジェクトを追加

▶リストの任意の位置へオブジェクトを追加

リストの任意の要素位置にオブジェクトを追加できます．**図14**で作成したlist1のリストを使って，**図15**のようにリストの先頭にbananaを追加します．ここではinsert関数を利用して，引数に要素番号，オブジェクトを指定します．

3行目でオブジェクトを追加しています．書式は，

```
オブジェクトの追加先のリスト.insert(追加先の要素番号，オブジェクト)
```

です．例では要素番号は0，オブジェクトとして文字列のbananaが格納されている変数text1を設定します．

4行目ではlist1をprint関数で画面出力します．2行目ではリストの先頭のオブジェクトはappleでしたが，以下ではbananaとなっています．

```
print(list1)
```

```
['apple', 'orange', 'lemon', 'banana']
```

```
list1.insert(0, text1)
print(list1)
```

3行目

```
['banana', 'apple', 'orange', 'lemon', 'banana']
```

図15　リストの実行例11…リストの任意の位置へオブジェクトを追加

▶リストの最後に別のリストを追加

演算子＋を使う以外にextend関数を使う方法が

あります．

図16の1～2行目で変数list1およびlist2にリストを代入しています．

3行目でextend関数を使ってlist1にlist2を追加しています．書式は，

追加先のリスト.extend(追加するリスト)

です．

4行目のprint関数で変数list1のリストを画面出力します．

```
list1 = ['apple', 'orange', 'lemon']
list2 = ['strawberry', 'kiwi']
list1.extend(list2)
print(list1)
```

```
['apple', 'orange', 'lemon', 'strawberry', 'kiwi']
```

図16　リストの実行例12…リストの最後に別のリストを追加

▶リストからオブジェクトを削除する

リスト中の特定の要素番号のオブジェクトを削除できます．ここでは図16のlist1の，4番目に入っているstrawberryを削除します．

図17の1行目で変数list1のリストを画面出力しています．

3行目で，pop関数の引数として削除したいオブジェクトの要素番号を指定します．リストの要素番号は0から始まる番号で，4つ目のstrawberryは要素番号が3です．削除が成功するとオブジェクトが画面出力されます．

5行目のprint関数で変数list1のリストを画面出力しています．

2行目でリストの最後にあったstrawberryが6行目で削除されています．

```
[2] print(list1)
```

1行目 `['apple', 'orange', 'lemon', 'strawberry', 'kiwi']`

```
[3] list1.pop(3)
```

```
'strawberry'
```

```
print(list1)
```

5行目 `['apple', 'orange', 'lemon', 'kiwi']`

図17　リストの実行例13…リストからオブジェクトを削除する

▶リストのオブジェクトを指定して削除

図18の1行目でリストのオブジェクトを画面出力しています．

3行目でremove関数の引数にorangeを指定して，変数list1のリストから削除します．remove関数ではリストの先頭から削除対象のオブジェクトを検索し，最初に見つかった一致するオブジェクトを削除します．削除対象と同じオブジェクトがリスト内に複数存在するとき，最初に見つかった同一のオブジェクトしか削除されません．

4行目のprint関数で変数list1のリストを画面出力しています．

```
print(list1)
```

```
['apple', 'strawberry', 'orange']
```

```
list1.remove('orange')
print(list1)
```

3行目

```
['apple', 'strawberry']
```

図18　リストの実行例14…リストのオブジェクトを指定して削除

▶リスト同士の比較

2つのリストに含まれるオブジェクトを比較演算子，

==

を使って比較できます．

図19の3行目のprint関数で比較の結果をブール型で画面表示しています．

リストの先頭から順にオブジェクトを比較し，結果として，画面にはFalseが表示されます．

オブジェクトの順序が異なっていても，同じオブジェクトが含まれている場合にTrueを返したいなら，5行目のようにset関数を使います．

変数list1およびlist2を引数に渡すとリストの要素番号に関係なく比較されます．

他にもsort関数を使ってリスト内のオブジェクトを並び替えてから比較する方法もあります．

```
list1 = [1, 2, 3]
list2 = [3, 1, 2]
print(list1 == list2)
```

```
False
```

```
print(set(list1) == set(list2))
```

```
True
```

図19　リストの実行例15…リスト同士の比較

5-3　タプル…書き換えも並び替えもできない要素群

表2　ネスト(多重)したタプルも作れる

書　式	説　明
()	空のタプル
(オブジェクト,)	単要素のタプル
(オブジェクト1，オブジェクト2，オブジェクト3)	複数要素のタプル
オブジェクト1，オブジェクト2，オブジェクト3	複数要素のタプル
(オブジェクト1，オブジェクト2，(オブジェクト3))	2重ネストされたタプル(タプルの中のタプル)

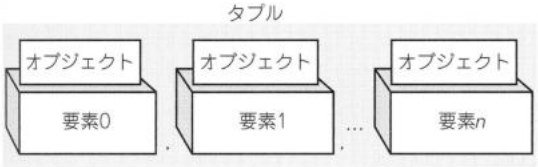


図20　複数のオブジェクトで作られる変更できないデータ構造

● タプルの利用目的

タプルはシーケンス型の1つです．タプルは，作成した後にオブジェクトを変更せずに利用したい場合に使われます(図20)．

プログラムを実行している間に変更されないオブジェクトを格納するのにタプルを使用します．リストと比べてタプルはメモリ消費が少ないので計算速度が速くなります．

例えば，都道府県名や生徒の氏名などのように，参照するものの，プログラムを実行している間は不変のオブジェクトを格納する場合に利用します．

その他にも，CSVファイルやJSONファイルに記述した各種パラメータを読み出し，タプルに格納して関数やクラスなどで参照するときにも役立ちます．各種パラメータを複数の変数に分けて格納すると，変数名が似ている場合など誤ってオブジェクトを更新してしまうことがあり，管理が大変になります．

タプルに格納しておけば，ソースコードの記述ミスがあっても，オブジェクトが書き変わってしまうことを防げます．

● リストとの違い

タプルはリストとは異なり，オブジェクトの追加・更新・削除ができません．タプルに格納するオブジェクトとして，変更されたくないパラメータ設定などによく利用されます．

● タプルの書式

表2に書式を示します．

タプルはシーケンス型の1つで，イテレータ・オブジェクトです．タプルを作るのは丸括弧ではなくカンマですので，丸括弧( )は省略できます．

1をタプルの要素として格納し，(1)のように記述するとオブジェクトの型はint型になりますが，(1,)のように記述するとtuple型になります．

タプルは，リストのようにappend関数，insert関数，extend関数，pop関数，remove関数などの操作はできません．タプルの内容を変えたい場合には再作成が必要です．

しかし，タプルもリストのようにスライスを使って特定のオブジェクトを参照できます．スライスは部分参照なので，オブジェクトの追加，更新，削除に当たりません．

2重ネストされたタプルや，タプルの中のタプルは丸括弧で区切ります．2重だけでなく，3重以上の多階層の配列のような構造を作成できます．

● 実行例

▶タプルの初期化

丸括弧を使って，空のタプルを作り，タプルを初期化します．図21のようにtype関数で変数tuple1のオブジェクトの型を調べるとtuple型となっています．

図21
タプルの実行例1…
タプルの初期化

```
tuple1 = ()
type(tuple1)
tuple
```

▶オブジェクトの入ったタプルを作る

図22の1行目ではオブジェクトと，(カンマ)でタプルを作り，変数tuple1に代入します．丸括弧を使わなくてもタプルになります．

2行目のようにtype関数でオブジェクトの型を調べるとtuple型となっています．

図22
タプルの実行例2…
オブジェクトの入ったタプルを作る

```
tuple1 = 1, 2, 3
type(tuple1)
tuple
```

▶要素の表示

一般的に丸括弧を付けてタプルを作ることが多いと思います．

図23の1行目でタプルを作成して変数tuple1に代入しています．タプルは複数オブジェクトをカンマで区切って作れますが，丸括弧を付けるとソースコードが読みやすくなります．

2行目で角括弧[]でタプルの要素番号を指定して，タプルの要素からスライスして特定のオブジェクトを参照します．リストと同じようにタプルの先頭から要素番号は0から始まります．

4行目のように要素の範囲を指定すると，タプルの一部を画面出力できます．

```
tuple1 = (1, 2, 3, 'A', 'B', 'C')
print(tuple1[4])
```

```
B
```

```
print(tuple1[1:4])
```

```
(2, 3, 'A')
```

図23
タプルの実行例3…要素の表示

▶多重になったタプル

図24の1行目で2重ネストされたタプルの例として，4つのオブジェクトを含むタプルを作り，変数tuple1に代入します．要素として3，4を持つタプルが最後のオブジェクトとして格納されています．

2行目のtype関数で変数tuple1に代入されたオブジェクトの型はtuple型となっています．

4行目で変数tuple1に格納されたオブジェクトをprint関数で表示しています．

```
tuple1 = 1, 2, (3, 4)
type(tuple1)
```

```
tuple
```

```
print(tuple1)
```

```
(1, 2, (3, 4))
```

図24
タプルの実行例4…多重になったタプル

5-4　キーと値をセットで格納する辞書

●辞書の利用目的

辞書（ディクショナリ）はリストのようにデータを格納する場合に利用します．リストやタプルのように値を格納でき，追加，更新，削除も可能です（図25）．

特徴はキーと値を組みとしたデータ構造であることです．格納されている値はキーにより参照でき，リストやタプルのように先頭から何番目に目的の要素があるかを知らなくても目的の値を参照，更新，削除できます．

● 辞書の書式

```
{}
{キー：値}
{キー：値1，キー：値2，キー：値3}
```

辞書はマッピング型のミュータブル・オブジェクトです．Pythonの組み込み標準型の中では，辞書だけがマッピング型です．

辞書は，"キー：値"対のカンマ区切りのリストを，波括弧でくくることで作成できます．辞書のキーとしてリストや辞書，その他のミュータブルな型は利用できませんが，ほぼ任意の型を使えます．値には，数字，文字列，リスト，タプル，辞書などを利用できます．

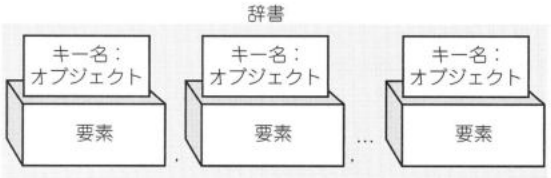


図25　キーと値（オブジェクト）を組み合わせたデータ構造

● 実行例

▶空の辞書を作る

空の辞書（ディクショナリ）は波括弧{}で作れます．図26のように空の辞書を変数dict1に格納し，print関数で画面表示すると値が空の辞書が表示されます．

```
dict1 = {}
print(dict1)
```

```
{}
```

図26
辞書の実行例1…空の辞書を作る

▶辞書にキーと値を設定

ここではキーに数字を使っていますが，文字列も使えます．コロンの後に半角スペースを挿入するとソースコードが読みやすくなります．

図27の1行目で辞書を作成して変数dict1に代入しました．見やすいようにキーと値を組み合わせ，3

つの組み合わせごとに改行しています．

2行目のprint関数でdict1の値を画面出力します．1行目でカンマの後に改行をしても画面出力は連続して出力されます．

```
dict1 = {
    1: 'apple', 2: 'orange', 3: 'lemon',
  4: 'banana', 5: 'pinapple', 6: 'grape',
  7: 'peach'
    }
print(dict1)
{1: 'apple', 2: 'orange', 3: 'lemon', 4: 'banana',
              5: 'pinapple', 6: 'grape', 7: 'peach'}
```

図27　辞書の実行例2…辞書にキーと値を設定

▶辞書のキーだけを表示

ここからは辞書がサポートする操作を説明します．

変数dict1に格納されている辞書から全てのキーだけを表示します．

図28の1行目のようにkeys()を使うと辞書に格納されているキーを返します．

3行目でキー数をlen関数でカウントします．

```
print(dict1.keys())
dict_keys([1, 2, 3, 4, 5, 6, 7])
print(len(dict1.keys()))
7
```

図28
辞書の実行例3…辞書のキーだけを表示

▶辞書の値だけを表示

図29の変数dict1の辞書に含まれる値だけを表示するには，values()を使います．その辞書が持つ全ての値が返されます．

```
print(dict1.values())
dict_values(['apple', 'orange',
'lemon', 'banana', 'pinapple',
'grape', 'peach'])
```

図29
辞書の実行例4…辞書の値だけを表示

▶キーと値の組みを表示

辞書のキーと値を表示するには，items()が利用できます．

図30の1行目のようにitems()を使うと辞書の内容が返されます．それをprint関数で画面出力しています．

3行目で辞書の内容からキーと値を参照して変数pairsに代入します．

4行目のtype関数で変数pairsのオブジェクトの型を確認しています．

6行目で変数pairsに格納されているオブジェクトをprint関数で画面出力します．

```
[3] print(dict1.items())
    dict_items([(1, 'apple'), (2, 'orange'), (3, 'lemon'),
    (4, 'banana'), (5, 'pinapple'), (6, 'grape'), (7, 'peach')])
[4] pairs = [(k, v) for (k, v) in dict1.items()]
[5] type(pairs)
    list
print(pairs)
    [(1, 'apple'), (2, 'orange'), (3, 'lemon'), (4, 'banana'),
    (5, 'pinapple'), (6, 'grape'), (7, 'peach')]
```

図30　辞書の実行例5…キーと値の組みを表示

▶値の削除

辞書から任意の値を削除するにはpopitem関数を利用します．削除対象は任意の値になり，どの値を削除するか指定できません．

図31の1行目で変数dict1に格納された辞書のpopitem関数を呼び出します．ここで任意の値が決まります．この関数は削除対象のキーと値を返します．変数aには削除対象のキーが，変数bには削除対象の値が代入されます．

このままでは辞書から何を返されたのか分からないので，2行目と4行目でそれぞれをprint関数で画面出力します．削除対象はキーが7，値がpeachとして出力されました．

6行目で辞書のキーと値を画面出力します．辞書の中から削除対象のキーと値がなくなっていることが分かります．

```
[7] a, b = dict1.popitem()
    print(a)
    7
[8] print(b)
    peach
print(dict1.items())
dict_items([(1, 'apple'), (2, 'orange'),
(3, 'lemon'), (4, 'banana'),
(5, 'pinapple'), (6, 'grape')])
```

図31　辞書の実行例6…値の削除

▶キーで値を参照する

キーを指定して辞書の中からキーと組みになった値を表示します．リストのようにスライスして特定の値を参照できます．

図32の例ではキーとして2を指定して，該当する値(orange)をprint関数で画面出力しています．

図32
辞書の実行例7…
キーで値を参照する

```
print(dict1[2])
```
```
orange
```

▶値を追加する

次にキーを指定して辞書に値を追加します．新規のキーとして8，値にbananaを辞書に追加します．

図33の1行目で変数dict1に格納された辞書にキーと値の組みを追加します．角括弧の中でキーの8を指定して値bananaを代入します．

2行目は追加結果の確認のため，items()で辞書からビュー・オブジェクトを返し，print関数で画面出力しています．

3行目を見ると，追加した値は辞書の最後に追加されています．

同一のキーが辞書にない場合，キーの値に関係なく見た目上は常に辞書の最後に追加されます．逆に辞書には要素番号がないため，辞書内の要素番号（位置）を指定してキーと値を挿入することができません．

```
dict1[8] = 'banana'
print(dict1.items())
```
```
dict_items([(1, 'apple'), (2, 'orange'),
(3, 'lemon'), (4, 'banana'), (5, 'pinapple'),
(6, 'grape'), (8, 'banana')])
```

図33　辞書の実行例8…値を追加する

▶値を更新する

辞書の値の更新は，更新対象の値のキーを指定して，新しい値を代入することで行います．

図34は辞書の既存キーの1に，新しい値としてpinappleを代入します．もともとキーの1と組みになる値はappleでしたが，3行目でupdate()の引数に波括弧で囲ったキーの1とpinappleを指定します．実行すると辞書の内容が更新され，変数dict1に更新済みの辞書が格納されます．

4行目で更新結果を確認するためprint関数で画面出力しています．このようにキー1の値がpinappleに更新されます．

```
dict1.update({1: 'pinapple'})
print(dict1.items())
```
```
dict_items([(1, 'pinapple'), (2, 'orange'),
(3, 'lemon'), (4, 'banana'), (5, 'pinapple'),
(6, 'grape'), (8, 'banana')])
```

図34　辞書の実行例9…値を更新する

▶特定の値を削除する（pop）

図35ではキーの3を指定して辞書から値を削除します．

1行目でpop()の引数に3を指定すると変数dict1に格納されている辞書からキーの3が検索され，該当の値を削除します．値と一緒にキーも削除されます．

2行目で削除処理後の辞書を確認します．print関数で辞書のキーと値を表示すると，キーの3とその値も消えています．

```
dict1.pop(3)
print(dict1.items())
```
```
dict_items([(1, 'pinapple'), (2, 'orange'),
(4, 'banana'), (5, 'pinapple'), (6, 'grape'),
(8, 'banana')])
```

図35　辞書の実行例10…特定の値を削除する（pop）

▶特定の値を削除する（del）

辞書の値を削除する方法としてdelも使えます．

図36の1行目で変数dict1を角括弧でキーに2を指定してスライスし，del文の引数にするとキーの2とその値が削除されます．

2行目で削除処理後の辞書を確認します．print関数で辞書のキーと値を表示すると，キーの2とその値も消えています．

```
del dict1[2]
print(dict1.items())
```
```
dict_items([(1, 'pinapple'), (4, 'banana'),
(5, 'pinapple'), (6, 'grape'), (8, 'banana')])
```

図36　辞書の実行例11…特定の値を削除する（del）

▶全てのキーと値を削除する

辞書の全ての値を削除して空の辞書を作成するにはclear()を利用します．値を削除するだけでなく，既存の辞書を初期化するときにも使えます．

図37の1行目のようにclear()を使うと変数dict1に辞書として格納された全てのキーと値を削除します．

2行目で変数dict1の辞書を確認すると何も格納されていないことが分かります．

図37
辞書の実行例12…全てのキーと値を削除する

```
dict1.clear()
print(dict1.items())
```
```
dict_items([])
```

さとう・せい

佐藤 聖

6-1 条件によって処理を分けるif文

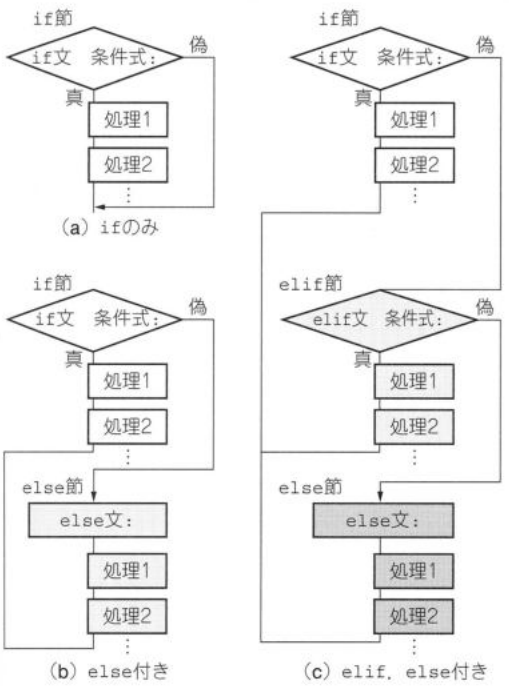


図1 条件式を評価してその後の処理を分岐する

リスト1 条件分岐の構文

```
if 条件式:
    条件式が真(True)のときに実行する処理                    ← 図1(a)

if 条件式:
    条件式が真(True)のときに実行する処理                    ← 図1(b)
else:
    条件式が偽(False)のときに実行する処理

if 条件式1:
    条件式1が真(True)のときに実行する処理                   ← 図1(c)
elif 条件式2:
    条件式2が真(True)のときに実行する処理
else:
    すべての条件式が偽(False)のときに実行する処理
```

● if文の利用目的

if文は条件分岐の処理に使われます(図1).

単独で利用する以外にも，for文やwhile文のようなループ処理でよく利用されます．条件分岐を使ってループから抜け出したり，ループの中で分岐によって異なる処理を実行したりするのに使われます．

● if文の書式

条件分岐の構文をリスト1に示します．

● 条件分岐の構文は3種類

if文の条件分岐は条件式の中で真偽を評価します．真のときにif節に記述された処理が実行され，偽のときには実行されません．

複数の条件分岐を行うには，if文の後に，elif文を追加して条件式を記述します．elif文は複数記述でき，条件式の真偽の評価はif文の条件式と同じです．elif節には条件式が真のときに実行させたい処理を記述します．

if文やelif文の条件式のいずれも偽として評価されたときに処理するのにelse文を使います．else文はif文やelif文の後に記述し，else節に処理を記述できます．

この3つの文を使った条件分岐には，ifだけ，if…else，if…elif…elseの記述パターンがあります．処理は上から順に実行されるので複数の条件式を使ったプログラムを書くときには，アルゴリズムに注意が必要です．条件式の記述に重複や漏れがないか，評価の順序に問題や矛盾がないかを，よく確認することが重要です．

elif文が大量に続くプログラムは，多数の条件式を処理するため処理に時間がかかります．複雑な条件式の組み合わせにならないよう，分割してシンプルなソースコードに見直すと記述ミスを減らせると思います．

● 実行例

▶ if節だけの場合

条件分岐で条件式の評価結果が真のときに処理する例です．if文の条件式で変数aの値と5を比較して，評価が真のときにprint関数が実行されるプログラムです．図2のプログラムでは，最初の行で変数aに5が代入されます．条件式の評価は真となり，画面にTrueが表示されます．もし，評価が偽ならif節の後に処理が接続され，if節の中にあるprint関数は実行されずに処理が終了します．

```
a = 5
if a == 5:
    print("True")
```

```
True
```

図2 if文1…if節だけ

▶ else節を使う場合

図3のプログラムは，条件式の評価が真と偽で異なる処理内容を実行したいときの例です．if文の条件

式で変数aの値と1とを比較するため評価は偽です．

そのため，if節にあるprint関数は処理されず，else節に処理が接続されます．

else文には条件式がありませんので，else節のprint関数が実行されて画面にFalseが表示されます．

もし，変数aに1を代入したら，2行目のif文の条件式で評価が真となり，if節のprint関数が実行され，画面にTrueが表示されます．処理はelse節の後に接続され，プログラムは終了します．

```
a = 5
if a == 1:
    print("True")
else:
    print("False")
```

```
False
```

図3
if文2…
else節を使う

▶複数の分岐を持つ条件分岐

図4は複数の条件式を使った条件分岐の例です．if文の条件式では，変数aの値と2とを比較しますが，変数aの値は5なので評価は偽になり，if節の後に処理が接続されます．

elif文の条件式では変数aの値と5を比較して評価が真になりますので，print関数が実行されてa is 5.が表示されます．

その後，else節の後に処理が接続されるため，print("etc.")は実行されずにプログラムは終了します．

もし，変数aが2のときは2行目のif文の条件式が真と評価され，if節のprint関数が実行されて，画面にa is 2.が表示されます．その後else節の処理が実行されて，プログラムは終了します．

```
a = 5
if a == 2:
    print("a is 2.")
elif a == 5:
    print("a is 5.")
else:
    print("etc.")
```

```
a is 5.
```

図4
if文3…条件
分岐

▶文字列の評価

条件式では文字列も真偽の評価に利用できます．**図5**の例では，変数aにappleを代入します．

2行目のif文では変数aとappleの文字列の比較で真として評価されます．直前の評価の真(True)と，その後に続くand Trueを組み合わせると，条件式はTrue and Trueと同じ意味です．

andでは，その前後のオブジェクトでブール演算が行われます．左辺の評価がTrueなので右辺のTrueが選択され，条件式全体の評価は真になります．

if節のprint関数が実行されて，Trueが画面表示されます．その後，処理対象がないのでプログラムは終了します．

```
a = "apple"
if a == "apple" and True:
    print("True")
```

```
True
```

図5
if文4…
文字列の評価

▶文字列とブール値の組み合わせ

図6はif文を比較して評価する例です．if文の条件式は，appleが代入された変数aと，文字列peachとを比較し，偽と評価されます．

その後，or文と比較対象にTrueがありますので，ここでの条件式はFalse or Trueと同じ意味です．

or文のブール演算では左側がFalseなので右側が選択されて，この評価は真になります．

if節のprint関数が実行されて，Trueが画面表示されます．その後，else節の後に処理が接続されてプログラムは終了します．

```
a = "apple"
if a == "peach" or True:
    print("True")
else:
    print("False")
```

```
True
```

図6　if文5…文字列とブール値の組み合わせ

6-2　ループから抜けるbreak文

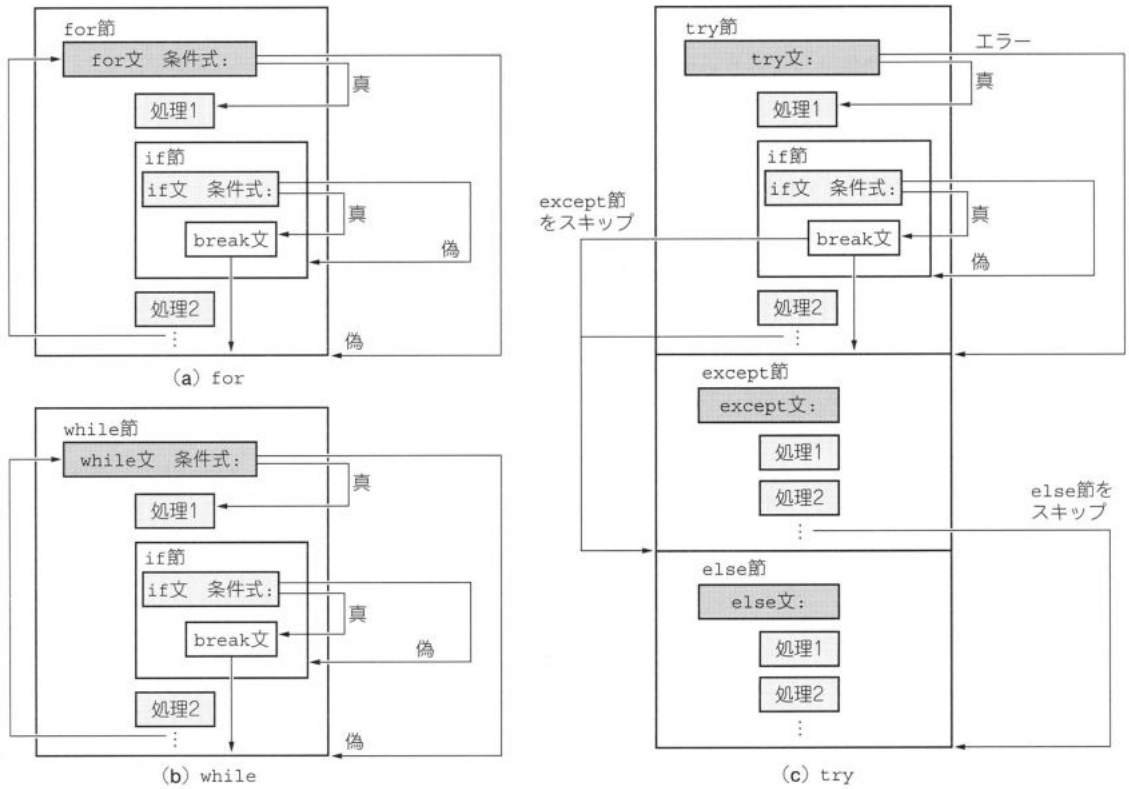


図7　プログラムの流れを制御するbreak文の使い方

●break文の利用目的

break文は，ループ処理やtry節の途中で，その節の処理から抜けるときに利用します．通常はif文と組み合わせて，特定の条件に該当したときにbreak文を実行します(図7)．

ただし，ループ内で定義する関数やクラスの中ではbreak文を利用できません．

ループの中でif文を使って処理を中断できるので，より複雑な条件で処理を制御できます．

●break文の書式

break文の書式は次になります．break文に引数はなく，単独で記述します．

```
break
```

●実行例

▶for文での使い方

for文のループの中でrange関数で作成された0～9までの数字を変数iに代入します(図8)．

if文の条件式で変数iが3であるかどうかを評価します．評価結果が真(True)のときにbreak文を実行してループから抜け出します．偽(False)のときはprint関数で変数iに代入された数値を画面に表示します．このプログラムを実行すると，以下のような結果になります．

```
for i in range(10):
    if i == 3:
        break
    print(i)
```

```
0
1
2
```

図8　break文1…for文での使い方

▶while文での使い方

図9のプログラムでは，変数numをカウンタとして利用します．変数numに0を代入してカウンタを初期化します．

2行目でwhile文の条件式にTrueを設定して無限ループを作ります．

3行目のif文で変数numのオブジェクトが5以上のときに真(True)になる条件式を設定しました．真のとき，if節のprint関数で変数numのオブジェクトが画面表示され，break文を実行してwhile文のループを抜け出し，while節の最後に接続され，プログラムが終了します．

3行目のif文の条件式が偽のときは，if節の最後に処理が接続され，6行目で変数numのオブジェクトに1を加算します．変数numに格納される値は，ループ処理で0～5まで変化し，6になったときにif文の条件式が真になるのでprint関数は6を画面表示します．次の行でbreak文が実行され，ループを抜け出してプログラムが終了します．

```
num = 0
while True:
    if num > 5:
        print(num)
        break
    num = num + 1
```

```
6
```

図9 break文2…while文での使い方

▶if文での使い方

図10のプログラムは，while…elseでループを実行します．

1行目で変数numをカウンタとして1を代入して初期化します．

2行目のwhile文の条件式は，変数numのオブジェクトが5以下のときが真となっています．この条件に当てはまる間は処理がループし，3行目のprint関数で変数numのオブジェクトを画面表示します．

4行目のif文で変数numのオブジェクトを2で割った余りが0のときに条件式が真となり，5行目のbreak文が実行され，プログラムが終了します．もし，4行目のif文の条件式の評価が偽だと，6行目で変数numのオブジェクトに1を加算してループを続けます．

通常，while文の条件式によりループが終了するときには，else節が実行されます．しかし，この例のようにbreak文を使ってループを抜け出すとelse節は実行されずに終了します．そのため，画面には1と2しか表示されません．

```
num = 1
while num <= 5:
    print(num)
    if (num % 2) == 0:
        break
    num = num + 1
else:
    print('stop')
```

```
1
2
```

図10 break文3…if文での使い方

▶try文での使い方

try文の中でbreak文を利用できます．except文，else文，finally文の処理に変わりはありません．図11の例ではtry節の中にfor文のループがあります．for文の条件式でrange関数により0～9のシーケンスが作成され，順に変数iに代入されます．

if文の条件式が偽のときは，5行目のprint関数で変数iのオブジェクトを画面出力します．

3行目のif文の条件式で変数iが5のときに評価が真になり，4行目のbreak文が実行されます．break文が実行されると，try節の途中であっても強制的にtry節の最後に接続されます．

このプログラムではtry節でエラーは発生せず終了したのでexcept節は実行されず，後続のelse節とfinally節は実行されます．処理結果は，以下の通りです．

```
try:
    for i in range(10):
        if i > 4:
            break
        print(i)
except:
    print("error")
else:
    print("success")
finally:
    print("end")
```

```
0
1
2
3
4
success
end
```

図11 break文4…try文での使い方

6-3　処理を指定した回数だけ繰り返すfor文

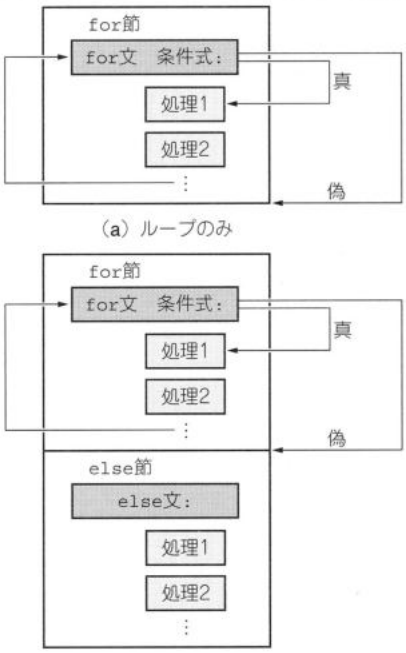


図12　処理を繰り返す回数が決まっている場合に便利

● for文の利用目的

for文は指定した回数を繰り返し処理するのに利用します．ループ処理ではリスト，タプルや辞書がよく利用され，要素や値を順番にアクセスしてループ処理します(図12)．

10回ループしたいときに，ループ回数をカウントしたり，range関数でシーケンスを作成したりして，for文の条件式に代入することで狙った回数だけ処理を繰り返すことも可能です．

● for文の書式

for文の書式は次のようになります．

```
for 変数 in イテラブルなオブジェクト:
    実行するプログラム
```

```
for 変数 in イテラブルなオブジェクト:
    実行するプログラム
else:
    ループ終了後に実行するプログラム
```

for文はシーケンス(文字列，タプルやリスト)や反復可能なオブジェクト(イテラブルなオブジェクト)内の要素に渡って反復処理をします．for文を使うとイテレータ(次の要素に繰り返しアクセスするインターフェース)から返された要素を条件式に代入してfor節の処理が実行され，全ての要素へアクセスを繰り返します．

for文の中で，else節を実行せずにループを終えるbreak文，ループの途中で残りの処理をスキップするのに利用するcontinue文を組み合わせることで，for文の条件式だけでなく，節中でif文と組み合わせてより複雑なループ処理を作れます．

● 実行例

▶単純なforループ

図13はfor文で4回のループ処理を行う例です．

1行目で変数list_numに数字のリストを代入しています．

2行目のfor文で条件式として変数list_numのリストの先頭から要素を取り出し，変数iに代入します．もし，リストから取り出す要素がなくなると条件式の評価は偽となってプログラムは終了します．

3行目のprint関数で変数iのオブジェクトを画面出力します．画面出力が終わると，2行目のfor文の変数list_numのリストから次の要素が変数iに代入され，3行目で画面出力します．これをリストの最後の要素となる4まで繰り返し，print関数で画面出力するため，画面には1～4が表示されます．

その後，2行目に戻り，for文でリストの中の次の要素を評価しようとしますが，変数list_numのリストに次の要素がないため，for文の条件式での評価は偽になり，ループから抜けてプログラムが終了します．

```
list_num = [1, 2, 3, 4]
for i in list_num:
    print(i)
```

```
1
2
3
4
```

図13　for文1…単純なもの

▶range関数でループ回数を設定

図14は1行目でfor文の条件式にrange関数を使って5回ループ処理を行う例です．このようにfor文の条件式に連番の数字を利用するときには，range関数で数のイミュータブルなシーケンスを作ると簡単です．

range関数に5を設定すると0～4までのシーケンスが作成されます．for文の条件式でシーケンスか

ら順に要素が変数iに代入され，for節の中のprint関数で変数iのオブジェクトが画面出力されます．

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

図14
for文2…range関数でループ回数を設定

▶ループ後に後処理

図15はfor文を使って5回のループ処理と後処理を行う例です．

上記のfor節にelse節を加えて，ループ後にprint関数でendを画面出力します．for節の処理後，後処理としてelse節に記述されたプログラムが実行されます．

```
for i in range(5):
    print(i)
else:
    print("end")
```

```
0
1
2
3
4
end
```

図15
for文3…ループ後に後処理

▶break文でループを抜ける場合の動作

図16はfor…else文からbreak文でループ処理を抜け出す例です．上記のプログラムの2行目にif文を追加したもので，変数iが2のときにbreak文が実行され，for節のループ処理から抜け出します．

1行目のfor文の条件式でrange関数では0～4のシーケンスを順に返します．

2行目のif文の条件式で要素が2のときにTrueとなり，if節のbreak文が実行されます．そうするとfor節のループを抜けて，5～6行目のelse節の後に処理がつながります．

画面出力は0，1だけが表示され，2，3，4とendが表示されません．このようにbreak文を使うとループ処理と後処理から抜け出すことができます．

```
for i in range(5):
    if i == 2:
        break
    print(i)
else:
    print("end")
```

```
0
1
```

図16
for文4…break文でループを抜ける

▶continue文の使い方

図17はif文とcontinue文を使ったループ処理の例です．continue文を使うと，その回のループをスキップして，次のループ処理を実行できます．for節やelse節の処理は継続されるので，条件に一致したときだけループの中で実行される処理をスキップするのに使われます．

1行目ではfor文の条件式でrange関数が返す0～4のシーケンスから順に取り出して変数iに代入しています．

2行目のif文で変数iのオブジェクトを2で割って余りが1のときに3行目のcontinue文が実行されます．もし，変数iのオブジェクトが1以外なら4行目のprint関数で変数iのオブジェクトを画面出力します．

余りが1のときは4行目のprint関数は処理されず，1行目のfor文に処理が移ります．その結果，for節では0，2，4が画面出力されます．

for節の処理が正常終了すると，6行目のprint関数でendを画面出力します．

```
for i in range(5):
    if i % 2 == 1:
        continue
    print(i)
else:
    print("end")
```

```
0
2
4
end
```

図17
for文5…continue文の使い方

6-4　一定条件が続く間ループ処理するwhile文

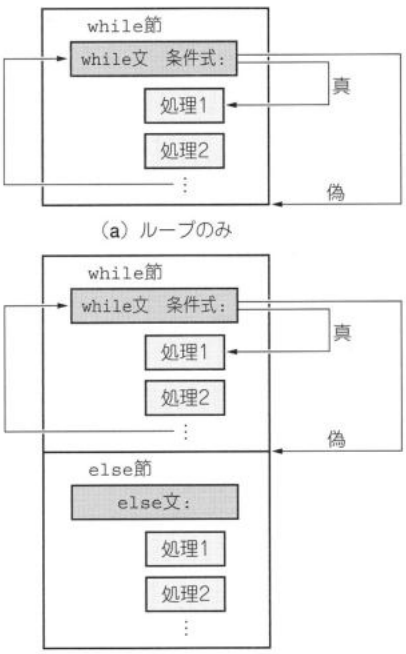


図18　条件を満たしている間だけループする

● while文の利用目的

while文は一定条件が続く間，繰り返し処理をするのに使われます(図18)．例えば，データが何行あるか分からないファイルから1行ずつ読み取る際に，全ての行を読み込みたいときには，for文よりwhile文の方が向いています．

while文はwhile節の実行をコンテキスト・マネージャによって定義されたメソッドでラップするために使われます．コンテキスト・マネージャは，with文の実行時にランタイム・コンテキストを定義するオブジェクトで，節(コード・ブロック)を実行するために必要な入り口および出口の処理を扱います．

コンテキスト・マネージャの代表的な使い方に，さまざまなグローバル情報の保存および更新，リソースのロックとアンロック，ファイルのオープンとクローズなどが挙げられます．

● while文の書式

書式をリスト2に示します．

● 条件式の構文は2種類

while文は条件式を評価して真(True)が返されたとき，while節に記述されたプログラムを繰り返し実行します．繰り返し処理の中で条件式の評価が偽を返したときにwhile節の処理を終了します．

else文を記述した場合，while文の条件式で評価が偽(False)を返したときにelse節に記述されたプログラムが実行されます．

また，条件式に評価が真を返し続けるプログラムを書くと，終わりのない無限ループになります．

リスト2　条件に合っている間，ループし続けるwhile文

```
while条件式:
    条件式が真(True)のときに実行するブロック内の処理

while条件式:
    条件式が真(True)のときに実行するブロック内の処理
else:
    条件式が偽(False)のときに実行するブロック内の処理
```

● 実行例

▶簡単なループ

図19の1行目で変数numに0を代入します．

2行目のwhile文の条件式では変数numが3未満のときにTrueを返し，ループ処理を行います．

3～4行目ではprint関数でTrueを画面出力し，変数numのオブジェクトに1を加算します．変数numのオブジェクトは，0～2のときにTrueが表示され，3のときに2行目のwhile文の条件式が返す値がFalseとなりwhile節の最後に処理が接続され，プログラムが終了します．

```
num = 0
while num < 3:
    print("True")
    num = num + 1
```

```
True
True
True
```

図19　while文1…単純なループ

▶else節付きのループ

図20はwhile文の条件式が真を返すときにTrueを，偽のときにFalseを画面表示します．

2行目のwhile文の条件式は，変数numのオブジェクトを2で割った余りを返し，それを0と比較して評価します．評価が真のときにはwhile節のプログラムが実行され，print関数でTrueが画面出力され，変数numのオブジェクトに1が加算されます．

もし，while文の条件式が偽を返したときにはelse文に処理が接続され，else節のprint関数

でFalseが画面出力されます．

else節は，while節が正常終了したときの後処理として機能します．

```
num = 0
while num % 2 == 0:
    print("True")
    num = num + 1
else:
    print("False")
```

```
True
False
```

図20　while文2…else付き

▶無限ループ

図21はwhile文の条件式がTrueであり続ければ無限ループができます．

具体的には1行目のように条件式にTrueを記述すると終わりのない繰り返し処理が行われます．while節のprint関数でTrueを画面に出力し続けます．

この例では入れていませんが，while節の中にif文でループを抜け出すための処理を用意して使うことが一般的です．6-2 ループから抜けるbreak文も参照してください．

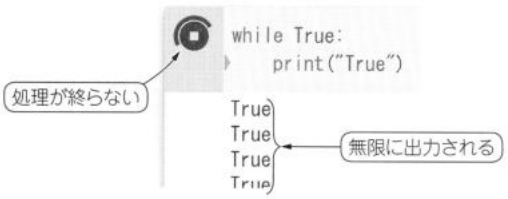


図21　while文3…無限ループ

6-5　エラー処理に使われるtry文

●try文の利用目的

try文は，Pythonプログラムでエラー発生時の対処に使われます(図22，次頁)．例えば，次のような処理で利用します．

- タイムアウト処理の考慮が必要なネットワーク通信を伴う処理(ウェブ・スクレイピング，データ通信，外部アプリケーションのAPI利用など)
- OS上のファイルやディレクトリなどの操作
- アプリケーションやOSコマンドをPythonプログラムから実行

これらの処理でエラーを起こさないような制御をPythonプログラムからはできない場合や，制御が困難な場合に，Pythonプログラム内に例外処理を記述します．

try文で外部コマンドやAPI，アプリケーションで発生したエラーに対応した例外処理を実行することで，Pythonプログラムが異常終了するのを防ぎます．

● 実行中に発生するエラーの例

文や式がPythonの構文的に正しい状態で，プログラムの実行中に検出されるエラーを例外と呼びます．

Pythonプログラムを実行するとソースコードの構文が解析され，処理が進む中で見つかります．

例えば，

- 0を用いた除算や剰余などの演算
- 存在しない変数の参照
- データ型が一致しないオブジェクト同士の演算

などがよくある例外です．エラー・メッセージでどの部分で何が起きたかが示されます．

3番目のデータ型が一致しないオブジェクト同士の演算は，例えば文字列の数字と整数のようにデータ型が一致しないオブジェクトの演算で発生する例外です．

図23に実際のプログラムを示します．

1行目の'2'が文字列，+演算子の後にある2が整数です．

この2つのオブジェクトは異なるデータ型なので，3行目で例外が発生した行番号，4行目にTypeError(型の不整合)があり，説明として「intオブジェクトを暗黙的にstrに変換できません」と示されています．

このような例外が発生するとプログラムが異常終了しますが，try文を利用して例外発生時の対応を記述することで処理を続行できるようにできます．

● try文の書式

書式をリスト3に示します．

```
'2' + 2
```

例外の発生するソースコード

```
TypeError                                 Traceback (mo
<ipython-input-1-d2b23a1db757> in <module>()
----> 1 '2' + 2

TypeError: can only concatenate str (not "int") to str
```

SEARCH STACK OVERFLOW

文字列同士しか接続できないと表示される

図23　例外発生時のエラー・メッセージ

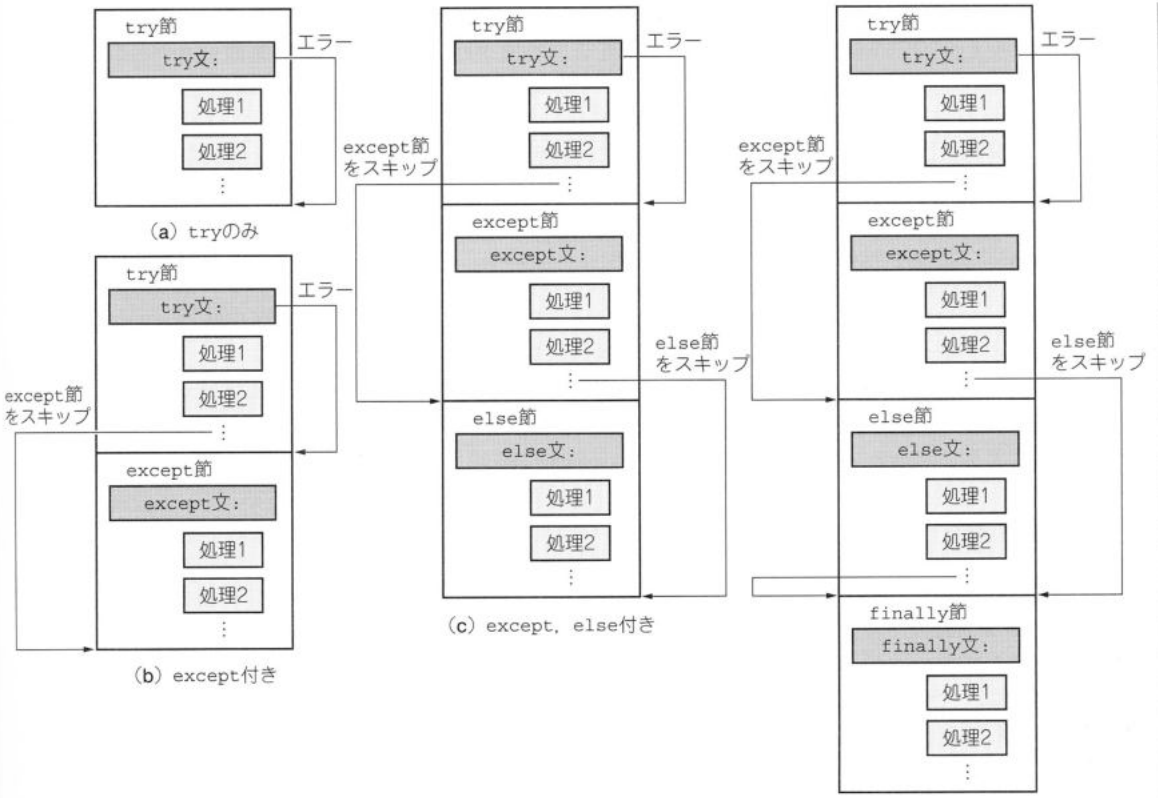


図22　実行時に例外が発生した場合に特定の処理を行わせる

リスト3　try文を使った例外処理の書式

try, except, else, finallyの組み合わせにより，これ以外のパターンも使える

```
try:
    try節のプログラム
```
← 図22 (a) のパターン

(a) tryのみ

```
try:
    try節のプログラム
except:
    try節で例外発生時に実行するexcept節のプログラム
```
← 図22 (b) のパターン

(b) except付き

```
try:
    try節のプログラム
except:
    try節で例外発生時に実行するexcept節のプログラム
else:
    try節が正常終了したときに実行するelse節のプログラム
```
← 図22 (c) のパターン

(c) except, else付き

```
try:
    try節のプログラム
except:
    try節で例外発生時に実行するexcept節のプログラム
else:
    try節が正常終了したときに実行するelse節のプログラム
finally:
    全ての節の処理後，常に実行するfinally節のプログラム
```
← 図22 (d) のパターン

(d) except, else, finally付き

try節に記述されたプログラムだけ例外処理が実行され，Pythonプログラム全体としてエラーで異常終了になりません．しかし，except節，else節，finally節でエラーが発生した場合，Pythonプログラムがエラーになり異常終了します．あらかじめエラーが発生しうるプログラムの箇所を絞り込んで，エラーが発生したときの検出や対処をプログラムに記述します．

▶try節

初めにtry文が記述されたtry節にあるプログラムが実行されます．try節でエラーによる例外が起きなければ，例外ハンドラは実行されません．一般的な例外ハンドラには構文エラー(SyntaxError)，例外(ZeroDivisionError, NameError, TypeErrorなど)があります．これ以外にもPythonに標準で定義されているエラーに組み込み例外があります．

例えば構文エラーの「SyntaxError: invalid syntax」を例にすると，SyntaxErrorは例外型，invalid

syntaxは関連値でどの動作が失敗したかを示す文字列で，プログラミングの問題をユーザに伝えます．例外はプログラムを記述してユーザ定義例外を作れます．プログラムの中で定義した例外，ライブラリやパッケージで定義された例外があります．

▶except節

try節の実行中に例外が発生すると例外ハンドラの検索が実行され，except節に接続されます．except文で1つ以上の例外ハンドラを指定でき，例外ハンドラの検索結果から得られた例外型の名(exceptキーワード)と一致する例外ならばexcept節のプログラムを実行します．もし，except文でexceptキーワードが指定されていない場合，全ての例外に対応します．

except節の実行後はtry節の最後に接続され，その後の処理を継続します．

▶else節

else文はtry節が正常終了したときに実行されます．通常はexcept文とセットで用い，except節の後に記述してtry節の正常終了時のプログラムとしてelse節，エラー時の処理としてexcept節に必要なプログラムを記述できます．try節でreturn文，break文，continue文のいずれかが実行されると，else節は実行されません．

▶finally節

finally節はtry節，except節，else節の最後に記述されます．通常は例外処理の後処理をfinally節に記述します．後処理ではないプログラムは，finally節に含めずにその後のプログラムに記述します．try節 でreturn文，break文，continue文のいずれかが実行されるとfinally節も実行されます．

try文，except文，else文，finally文を使った記述パターンは以下の通りになります．

- tryだけ
- try…except
- try…finally
- try…except…finally
- try…except…else…finally

● 実行例

▶単純な例外処理

図24のプログラムではtry文に処理したい内容，except文にエラー発生時の例外対応処理を記述しました．

1行目のtry文と2行目のtry節でprint関数の引数内の計算が行われ，画面に101を出力します．

try節は正常終了すると，3～4行目のexcept文とprint関数は実行されません．

もし，3行目でエラーが発生するとexcept節が実行され，プログラムは正常終了します．

このように，try節の処理がエラーになってもexcept節を用意するとエラー時の例外処理を用意でき，さらにプログラム全体がエラーになることを防げます．

```
try:
    print(100 + 1)
except:
    print("error")
```

```
101
```

図24　try文1…単純な例外処理

▶else節，finally節を追加した例外処理

図24のプログラムにelse節，finally節を追加したのが**図25**のプログラムです．

この例では1～2行目のtry文を含むtry節のプログラムは101を画面出力して正常終了します．

try節でエラーが発生していないので，3～4行目のexcept節はスキップされ，4行目のprint関数が実行されず，errorは画面出力されていません．

1～2行目のtry節が正常終了しているので，5～6行目のelse節が実行され，end this processが画面出力されています．

最後に後処理として7～8行目のfinally節が実行され，last processを画面出力します．

このように，

- 目的の処理
- 例外処理
- 正常終了時の処理
- 後処理

を項目を分けて記述できるため，ソースコードの可読性が良くなります．

```
try:
    print(100 + 1)
except:
    print("error")
else:
    print("end this process")
finally:
    print("last process")
```

```
101
end this process
last process
```

図25　try文2…else節やfinally節を追加した例外処理

▶例外発生時の流れ

エラー発生時の例外処理プログラムがどのように制御され，except節が実行されるか確認します．

まず，エラーになるプログラムを実行してみます．図26の処理ではprint関数の引数にint型のオブジェクトの100とstr型のオブジェクトのabcを演算子＋の両辺に置いています．オブジェクト型が異なっているので演算も結合もできずエラーが発生します．

```
print(100 + "abc")
```

```
---------------------------------------------------------------------------
TypeError
<ipython-input-21-e8cde82f2522> in <module>
----> 1 print(100 + "abc")

TypeError: unsupported operand type(s) for
```

SEARCH STACK OVERFLOW　エラーが発生する

図26　エラー発生時

次にtry節に上記のprint関数を記述して実行します．

図27の1～2行目のtry節のprint関数はエラーになるため，処理は3行目のexcept文に移り，4行目のprint関数が実行されて画面にerrorが出力されます．上記のprint関数の処理ではエラーになり処理が止まりました．

しかし，try文，except文を使用すると処理全体として停止せずに処理が継続されます．

except文には引数として特定のエラーを指定できますが，指定しなかった場合は全てのエラーが対象になります．

```
try:
    print(100 + "abc")
except:
    print("error")
```

```
error
```

図27
try文3…例外発生時の処理

▶特定のエラーだけを処理

図28のようにexcept文の引数にTypeErrorを指定すると，このタイプのエラーのときだけ，例外処理を実行できます．

なお，except文の引数に指定できるのは，組み込み例外のクラス階層やユーザ定義例外です．

ここでは3行目でexcept文の引数にTypeErrorを，2行目のprint関数で返されたエラーをその後利用するために，as Eを指定してエラーに別名を付けました．この別名はexcept節で利用でき，4行目のようにprint関数の引数に使い画面出力します．

```
try:
    print(100 + "abc")
except TypeError as E:
    print("catch TypeError: ", E)
```

```
catch TypeError:  unsupported operand type(s) for +: 'int' and 'str'
```

図28　try文4…特定のエラーだけを処理

▶特定のエラー（複数）だけを処理

except文の引数に複数の組み込み例外のクラス階層にあるエラーをまとめて指定できます（図29）．

3行目のようにexcept文の引数に複数のエラーを丸括弧で囲い，カンマで区切って指定できます．

いずれのエラーも別名のEに設定できるので，4行目のprint関数の引数に設定して画面出力できます．

```
try:
    print(100 + "abc")
except (SyntaxError, TypeError) as E:
    print("error: ", E)
```

```
error:  unsupported operand type(s) for +: 'int' and 'str'
```

図29　try文5…複数の特定エラーを処理

▶発生したエラーによって例外処理を分ける

複数のexcept文を記述することもできます．エラーの種類によって例外処理を変えるときに使います．例外処理も意味のまとまりで整理するとソースコードの可読性が良くなります．

図30の1～4行目は上記の例と同じ内容です．5～6行目に新たにexcept節で例外処理を追加しました．

特にOSErrorに関連する例外の種類はさまざまですし，Python外部ライブラリには独自のユーザ定義例外が設定されています．例外の発生が組み込み関数，標準ライブラリや外部ライブラリなど複数の例外に対応できるようにするには，複数のexcept文に整理して記述するとソースコードが読みやすくなります．

```
try:
    print(100 + "abc")
except TypeError as E:
    print("catch TypeError: ", E)
except ZeroDivisionError as E:
    print("catch ZeroDivisionError: ", E)
```

```
catch TypeError:  unsupported operand type(s) for +: 'int' and 'str'
```

図30　try文6…エラーによって例外処理を分ける

◆参考文献◆

(1) Pythonの組み込み例外．
https://docs.python.org/ja/3.7/library/exceptions.html#bltin-exceptions

さとう・せい

佐藤 聖

7-1 繰り返し利用する機能を関数としてまとめるdef文

● def文の利用目的

def文を用いるとユーザ定義関数オブジェクトを定義できます．一般的に繰り返し利用する機能を関数としてまとめたり，プログラムの可読性を良くするために機能を整理して関数にまとめたりします．

関数を定義するとプログラムの中でローカル関数として呼び出せます．同じアルゴリズムが繰り返し出てくるとソースコードは煩雑になりますが，関数にまとめるとプログラム全体を読みやすく整理できます．

● def文の書式

def文の書式は次のようになります．

▶関数の定義その1

```
def 関数名():
    実行するプログラム
```

▶定義した関数の呼び出し

```
関数名()
```

▶関数の定義その2

```
def 関数名(ローカル変数のオブジェクト，ローカル変数のオブジェクト):
    実行するプログラム
```

▶定義した関数の呼び出し

```
関数名(オブジェクト，オブジェクト)
```

def文で定義したユーザ定義関数は，プログラムの中で関数名を使って呼び出せます．必要に応じて関数の引数を設定でき，関数が受け取ったオブジェクトはローカル変数として関数内で利用できます．関数名に引数を指定して呼び出します．

また，関数から返される値は，最初に呼び出した関数名に返ってきます．値を再利用するには変数に格納して，その後の処理で利用します．関数が別の関数を呼び出すことで入れ子構造にでき，関数間で値を連携できます．

なお，関数が定義できるように，関連する複数の関数を集めてユーザ定義クラスを作れます．そのためにはクラスを作るときに，関連する小さな機能のまとまりを関数として整理するとよいでしょう．

● 実行例

▶簡単な関数

図1のプログラムでは，関数functionを作成し，その中でprint関数を記述しています．

3行目のようにfunction()を実行すると，def文で定義した関数functionを実行します．関数内のprint関数でabcを画面表示します．この形は最も単純なパターンです．

```
def function():
    print("abc")

function()
```

```
abc
```

図1 def文1…簡単な関数

▶関数内での複数の関数呼び出し

例えば，def文で定義した関数の中に複数の関数を記述して，通常のプログラムのように実行できます．

この関数を使用するときは，図2の4行目のようにプログラムの中でfunction()と記述するだけで実行できます．

同じアルゴリズムを何度も使用するときに利用価値が高く，プログラムを読みやすく簡単に記述できます．

```
def function():
    for i in range(5):
        print(i)

function()
```

```
0
1
2
3
4
```

図2 def文2…関数内での複数の関数呼び出し

▶関数内で定義した変数は関数内だけで使える

ただし，上記で定義した関数内で宣言された変数iのオブジェクトは，関数functionの中でしか利用できないローカル変数です．

試しに，上記の関数の外でprint関数を使って変数iに格納されているオブジェクトを画面表示しようとすると，変数iが見つからずにNameErrorになります(図3)．

```
print(i)
---------------------------------
NameError
<ipython-input-1-3b184248ad5a> in
----> 1 print(i)
NameError: name 'i' is not defined
```


(このiが見つからないため)

(エラーが発生)

図3 関数内で定義した変数は関数内だけで使える

▶関数にオブジェクトを渡す

定義した関数内にある変数iのオブジェクトは，暗

黙的には関数呼び出し元のプログラムに伝達されません．上で見たように，関数内で定義した変数はローカル変数として振るまいます．

関数からプログラムにオブジェクトを渡すためには，後ほど紹介するreturn文を使います．

上記とは別に，プログラムから関数を呼び出す際に，何らかのオブジェクトを関数に渡したいこともあるでしょう．その場合，def文の中で関数名の引数としてローカル変数を定義します．ユーザ定義関数内のローカル変数の定義は，通常の変数と同じように代入するオブジェクトによってオブジェクト型が自動的に決まります．

図4の例では，関数functionの引数にローカル変数xとyを定義し，その変数のオブジェクトを使ってprint関数の中で加算して結果を画面出力します．

プログラムで関数を呼び出す際に，関数名の引数として10と1を設定すると，def文で定義した関数functionにオブジェクトが伝達され，オブジェクトを処理に利用できます．

```
def function(x, y):
    print(x + y)

function(10, 1)
11
```

図4
def文3…関数にオブジェクトを渡す

▶関数に変数を渡す

関数を呼び出す際に引数として，オブジェクトだけでなく変数，リスト，タプル，辞書なども渡せます．

ここでは変数aのオブジェクトをfunction(a)のように記述して関数に渡します．関数内ではローカル変数textにオブジェクトが代入され，print関数で画面出力します．呼び出し元の変数aと関数のローカル変数textのように変数名を分けておくと混乱が少ないです(**図5**)．

```
a = 'abc'
def function(text):
    print(text)

function(a)
abc
```

図5
def文4…関数に変数を渡す

▶グローバル変数

ユーザ定義関数からオブジェクトを返す方法として，グローバル変数があります．

しかし，プログラム全体から参照できるグローバル変数を利用するのはお勧めできません．

図6のプログラムでは1行目で変数aにabcを代入し，3行目の通り，関数functionの中でglobal文を使い，変数aをグローバル変数として宣言しています．

そうすると変数aはローカル変数にならず，関数の外でも同じ変数を利用できますが，アルゴリズムが複雑になり扱いづらくなる場合があります．

4行目のprint関数ではグローバル変数aに格納されたオブジェクトを画面出力します．

その後，5行目で変数aに対して数値の123を代入しました．

関数の処理が終わり，8行目のprint関数で変数aのオブジェクトを画面に出力すると，関数内で代入していた123が表示されます．

このようにglobal文を使うとプログラム全体で同じ変数名を利用できます．

```
[1]  a = 'abc'
     def function():
         global a
         print(a)
         a = 123

[2]  function()
     abc
     print(a) ←(8行目)
     123
```

図6
def文5…グローバル変数

グローバル変数を使うと関数の内外で同じ変数が利用できますが，関数内での利用後も変数に代入されたオブジェクトが残っているため注意が必要です．どこでどんなオブジェクトが代入されたかを見つけにくくなります．

例えば，グローバル変数の変数名を変更しなければならないときに，その修正が広範囲になる可能性があります．プログラムの修正を最小限の範囲にとどめるには，def文で定義するユーザ定義関数の中では，ローカル変数として定義した方がよいでしょう．

関数やクラスを定義するような大きなプログラムを記述するときには，プログラムに潜在的な不具合のある可能性が高まります．小さな機能のまとまりで関数を定義し，修正が簡単になるような記述をするのがお勧めです．

7-2　呼び出し元へ制御を戻すreturn文

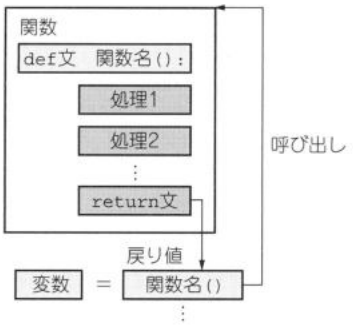


図7　複数回行う処理は関数にしておくと便利

● return文の利用目的

return文は，def文で定義した関数から呼び出し元に戻るときや，呼び出し元へ値を返すときに利用します(図7)．

try文から外に抜け出すときにも利用され，そのときはfinally節も実行されます．

いずれの文でもif文の条件式によってreturn文を実行して節から抜けるといった使い方以外に，条件分岐を使ってreturn文で異なる戻り値を関数の呼び出し元に返すことができます．関数から返す値がないときは，return文の戻り値を省略することもできます．

● return文の書式

return文の書式は次のようになります．

▶関数の定義(戻り値なし)
```
def function():
  return
```
▶関数の呼び出し
```
function()
```
▶関数の定義(戻り値あり)
```
def function():
  return 0
```
▶関数の呼び出し(戻り値を変数に代入)
```
x = function()
```

return文はユーザ定義関数から値を返すときに利用します．return文を使うと，式リスト(またはNone)を戻り値として，現在の関数呼び出しから抜け出します．式リストが省略されているときにはNoneに置き換えられます．

イテレータを戻り値とするジェネレータ関数は，イテレータ・オブジェクトでもあります．return文はジェネレータ(イテレータの一種であり，1要素を取り出そうとするたびに処理を行う)の終わりを示し，StopIteration例外を送出させます．返された値があれば，StopIterationを構成する引数に使われ，StopIteration.value属性です．返された値は同じオブジェクト型を取るオブジェクトとして，変数，リスト，タプル，辞書などに代入できます．

非同期ジェネレータ関数(戻り値のない関数)では，引数なし(式リストなし)のreturn文は非同期ジェネレータの終わりを示し，StopAsyncIterationを送出させます．

● 実行例

▶戻り値のない場合

return文の引数を定義しないときには，Noneを指定したものとして処理されます．def文の中でprint関数で文字列abcを画面出力し，return文で呼び出し元のfunction()に返されます(図8)．

return文には引数がありませんので，戻り値としてNoneが返され，変数aに代入されます．

最後にprint関数で変数aに代入されているオブジェクトを画面出力するとNoneが表示されています．

```
[1]  def function():
         print('abc')
         return

[2]  a = function()
     abc
     print(a)
     None
```

図8　return文1… 戻り値のない場合

▶戻り値を指定する場合

return文に引数を定義すると関数の実行後，引数の値を返せます．

ここでは，先ほどと同じようにa = function()で関数を呼び出し，戻り値を変数aに代入します．print関数で文字列abcを画面表示します(図9)．

3行目のreturn文でdefを引数として，4行目で関数から戻り値として受け取り，オブジェクトを変数aに代入しています．

6行目のprint関数で変数aのオブジェクトを画面出力します．関数内で戻り値としたdefが表示され，正常に関数が実行されます．

```
[1]  def function():
         print('abc')
         return 'def'

[2]  a = function()
     abc

     print(a)
     def
```

図9
return文2…戻り値を指定する場合

▶複数の戻り値から選択

関数の中でif文で条件に応じて，return文の引数を指定すれば狙った戻り値を返せます．

図10のプログラムでは，3行目のif文の条件式で変数aに格納されているabcにzが含まれているかどうかを判定します．ここでは含まれていないため評価結果は偽(False)です．

5～6行目のelse節が実行されます．

7行目で関数の戻り値は変数aに格納され，print関数でnoが画面に表示されます．

```
def function():
    a = 'abc'
    if 'z' in a:
        return 'yes'
    else:
        return 'no'

a = function()  ←7行目
print(a)
no
```

図10
return文3…複数の戻り値から選択

▶複数の戻り値を渡せる

return文の引数には複数の戻り値を指定できます．図11の例では，関数の中で変数aに123，変数bにabcを代入して，4行目のreturn文の引数にそれぞれの変数を設定しています．

5行目で関数の戻り値を受け取り，変数aのオブジェクトは変数dに，変数bのオブジェクトは変数eに代入しています．return文の引数の順で値を受け取ります．

6行目のprint関数で変数dのオブジェクトを，8行目のprint関数で変数eのオブジェクトを画面出力します．

このようにreturn文の引数には複数の戻り値を渡すことができます．

```
[1]  def function():
         a = 123
         b = 'abc'
         return a, b

[2]  d, e = function()
     print(d)
     123

     print(e)
     abc
```

図11
return文4…複数の戻り値を渡せる

▶リストを戻り値にする

図12の例は，4行目でreturn文の引数にリストを指定しています．6行目で関数から返された値をprint関数で画面出力します．関数内で作ったリストをそのまま呼び出し元に返すことができます．

```
def function():
    a = 123
    b = 'abc'
    return [a, b]  ←4行目

new_list = function()
print(new_list)  ←6行目
[123, 'abc']
```

図12
return文5…リストを戻り値にする

▶タプルを戻り値にする

図13はタプルを関数の戻り値として設定したケースです．

```
def function():
    a = 123
    b = 'abc'
    return (a, b)

new_tuple = function()
print(new_tuple)
(123, 'abc')
```

図13
return文6…タプルを戻り値にする

▶辞書を戻り値にする

図14は辞書を関数の戻り値として設定したケースです．

```
def function():
    a = 123
    b = 'abc'
    return {1:a, 2:b}

new_dict = function()
print(new_dict)
{1: 123, 2: 'abc'}
```

図14
return文7…辞書を戻り値にする

さとう・せい

佐藤 聖

8-1 ファイル操作の準備open関数/close関数

表1 ファイルをオープンするときのモード

文字	意味
r	読み出し用に開く(省略時のデフォルト)
w	新規作成，ファイルが存在する場合は上書き
x	新規作成，書き込み用でファイルを開き，ファイルが存在する場合は失敗
a	書き込み用に開き，ファイルが存在する場合は末尾に追記

(a) 読み出し/書き込みの指定

文字	意味
b	バイナリ・モード
t	テキスト・モード(省略時のデフォルト)

(b) テキスト/バイナリの指定

文字	意味
+	既存ファイルの上書き

(c) 上書きの指定

プログラムの中で行うファイル操作とは，OSが管理するファイル・システム上に存在するバイナリ・ファイル，テキスト・ファイル，画像ファイルなどにアクセスして読み書きする処理です．

Pythonプログラムから直接ファイルを操作できませんので，ファイル・システムを通じてファイルのオープン，クローズ，読み書きを行います．

具体的にはPythonの関数を使って，プログラムとファイル・システム間でストリームと呼ばれる論理インターフェース作ってファイルを操作します．ファイルに対するプログラムとファイル・システム間のストリーム入出力操作が，OS上から見たファイル操作となります．

● ファイルのオープンとクローズ

ファイルを読み書きするためにまずはファイルをオープンします．このためのopen関数は，デフォルトではテキスト・モードでファイルを開きます．

特定のエンコーディングでエンコードされたファイルに対して文字列を読み書きできます．エンコーディングは，Windows，Linux，masOSなどのプラットフォームに依存します．テキスト・モードの読み取りでは，これらのプラットフォーム固有の行末記号が使われます．

バイナリ・データが記録されているファイルはバイナリ・モードで開くことができ，読み書きも可能です．

ファイルの利用が終わったらclose関数で閉じて，ファイルを開放します．

● open関数/close関数の書式

▶open関数

```
open(ファイル名, mode='r', buffering=-1,
encoding=None, errors=None,
newline=None, closefd=True,
opener=None)
```

▶close関数

```
close()
```

▶open関数の書式解説

引数でよく利用されるのが，mode，encoding，newlineです．buffering，errors，closefd，openerについては，通常はデフォルトのまま使うと思います．

・ファイル名

open関数に渡すファイル名は，ファイルが保存されているディレクトリやフォルダを含めたパスを含めて指定できます．例えば，piユーザのホーム・ディレクトリにファイルが置かれている場合，

```
/home/pi/ファイル名
```

と記述できます．

・mode

ファイルが開かれるモードを指定します．省略した場合のデフォルトは"r"になり，読み出し用にテキスト・モードで開きます．モードの一覧を**表1**に示します．

・encoding

ファイルのエンコードやデコードに使われるtext encodingの名前です．このオプションはテキスト・モードでだけ使用できます．デフォルト・エンコーディングはプラットフォーム依存(locale.getpreferredencoding()で調べられる)ですが，Pythonでサポートされているエンコーディングはどれでも使えます．

日本語環境であれば，cp932，euc_jp，shift_jis，utf_8がよく利用されます．

読み書きしようとするファイルのエンコードやデコードに合わせて指定できます．

・newline

ユニバーサル改行モードの動作を制御し，テキスト・モードでだけ動作します．デフォルトではNoneになっており，その他に，

```
'', '¥n', '¥r', '¥r¥n'
```

のいずれかを指定できます．

入出力ではデータを小さい単位が連続したものと捉えて，データの入出力を流れとして抽象化したストリームとして扱います．

データは上流から下流へ流れるものと見なして，データの入出力や送受信を滞留させずに低遅延で取り扱う形態です．データ送信が終わったら蛇口を閉じて水を止めるように，ストリームをフラッシュすることでデータの流れを止めます．

例えば，動画ストリーミング配信であるときは継続的に，場合によっては断続的にデータ送受信されるので，都度ストリームをフラッシュせず，まとめてストリームを閉じることがあります．閉じることでデータを一時的に蓄積するメモリ領域（メモリ・バッファ）が開放されます．

ストリームから入力する場合，newlineの指定値がNoneまたは省略されているとき，改行コードに自動変換されます．それ以外の指定値のときは自動変換しません．ストリームに出力するときは，指定に従い改行コードを含めて文字列に変換します．

▶close関数の書式解説

ストリームをフラッシュして閉じます．引数はありません．

● 実行例

ここではテスト用のファイルとして，テキスト形式のtest.txt（**リスト1**）とバイナリ形式のtest.dat（**リスト2**）を用意しました．バイナリ・ファイルは，

```
pi@raspberrypi:~ $ echo -en "¥xff¥xff¥xff¥xff" > test.dat
```

を実行して作りました．

Pythonプログラムからtest.txtを開いて，ファイル・タイプとファイルの中身を表示させます．

▶テキスト・モードの場合

図1の1行目でファイル名を変数pathに代入し，2行目のopen関数でファイルを開いて，変数fにファイル・オブジェクトを格納します．ここではデフォルトのテキスト・モードでファイルが開かれます．

3行目のread関数で変数fのファイル・オブジェクトが文字列として返され変数sに代入されます．

4行目でファイルを閉じます．ここまでがファイルのオープン，読み出し，クローズになります．基本的にファイルを開いたら，使用後に閉じてファイルをプログラムから解放する必要があります．

リスト1　ファイルの読み書きに使用するテキスト・ファイル

```
line 1
line 2
line 3
line 4
line 5
```

リスト2　ファイルの読み書きに使用するバイナリ・ファイル

```
00000000: ff ff ff ff
```

5行目で変数sに格納されたオブジェクトの種類を調べるため，type関数に代入します．6行目がprint関数でオブジェクトの種類（ファイル・タイプ）を出力した結果です．

7行目のprint関数で，変数sに格納されている文字列を以降の行に表示しています．

図1　ファイル操作の準備1…テキスト・モード

▶バイナリ・モードの場合

図2の1行目でtest.datを変数pathに代入します．

3行目のopen関数のmode引数にバイナリ・モードで読み出すようbrを設定しています．

その他の行は，先ほどのテキスト・ファイルと同様です．

7行目にファイル・タイプが表示されています．bytesと表示されており，バイナリ・データだと分かります．

9行目の画面出力も最初にbが表示されており，バイナリ・データであることが分かります．

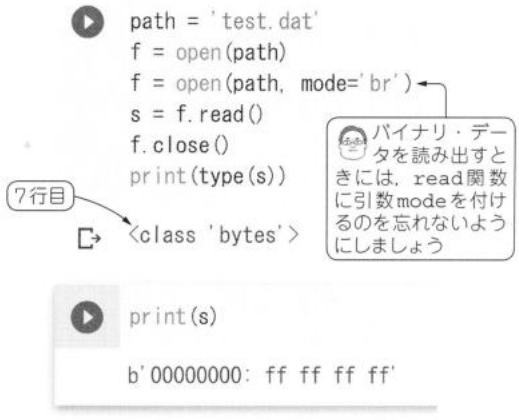


図2　ファイル操作の準備2…バイナリ・モード

8-2　ファイル読み出し関数 read関数/readlines関数/readline関数

● 利用目的

テキスト・ファイルやバイナリ・ファイルの読み出しには，ファイル全体を読み出すread関数，ファイル全体をリストとして読み出すreadlines関数，ファイルを1行ずつ読み出すreadline関数を利用します．

これらは，I/Oストリームからデータを読み出すためのAPIを提供します．ファイルの内容を取得して，Pythonプログラムで利用できます．

● ファイル読み出し関数の書式

▶指定したサイズを読み出す

```
read(size)
```

▶全部の行を読み出す

```
readlines()
```

▶1行だけ読み出す

```
readline(size)
```

● ファイル読み出し関数の解説

▶read関数

read関数のsizeに指定された数字は，テキスト・モードでは文字列，バイナリ・モードではバイトとして読み出されます．この値を省略したり，負の数だったりするときはファイルの内容全てを読み出します．

▶readlines関数

ファイルの全ての行をリスト形式で読み出します．

▶readline関数

readline関数は1行だけ読み出します．改行文字(改行コード)は読み出された文字列の終端に残して出力します．

sizeを省略したり，負の数だった場合，行全体または行の残りの部分が読み出されます．

ファイル全体を行単位で読み出すときには，for文と組み合わせて1行ずつ読み出すような使い方をします．

ファイルの終端に達すると空の文字列''を返します．

● 実行例

ファイルを開いた後，幾つかの読み出し方法でファイルの中身を読み出してみます．

▶テキスト・モード＋read()

test.txtを開いて，ファイル・タイプとファイルの中身を表示させます．

図3の1行目でファイル名を変数pathに代入し，2行目のopen関数でファイルを開いて変数fにファイル・オブジェクトを格納します．ここではデフォルトのテキスト・モードでファイルが開かれます．

3行目のread関数で変数fのファイル・オブジェクトが文字列として返され変数sに代入されます．

4行目でファイルを閉じてファイルを解放します．

5行目のprint関数で変数sに格納されている文字列を表示しています．

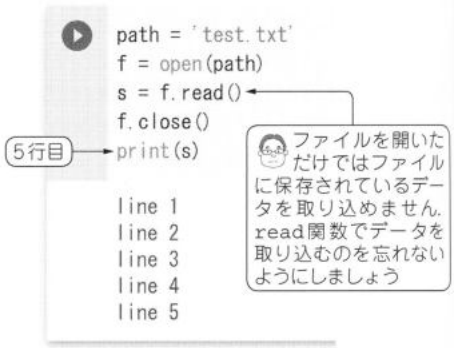


図3　ファイル読み出し関数1…テキスト・モード＋read()

▶テキスト・モード＋readlines()

ファイルを開き，変数fにファイル・オブジェクトを格納するところまでは同じです．

図4の3行目のreadlines関数で，変数fのファイル・オブジェクトが文字列として返され，変数lに代入されます．

4行目でファイルを閉じてファイルを解放します．

5行目のprint関数で変数lに格納されている文字列を以降の行に表示しています．

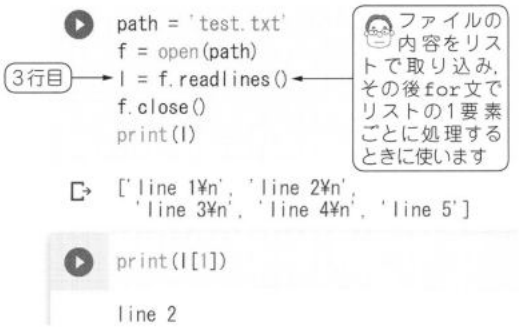


図4　ファイル読み出し関数2…テキスト・モード＋readlines()

▶テキスト・モード＋readline()

ファイルを開き，変数fにファイル・オブジェクトを格納するところまでは同じです．

図5の3行目のreadline関数で，変数fのファイル・オブジェクトの1行分を文字列として返して，

変数single_lineに代入します．

4行目のprint関数で変数single_lineに格納されている文字列を以降の行に表示しています．

6行目でファイルを閉じてファイルを解放します．

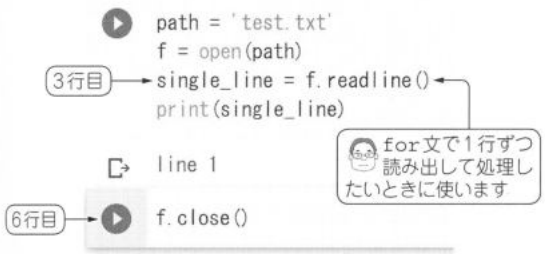


図5　ファイル読み出し関数3…テキスト・モード＋readline()

▶バイナリ・モード＋read()

test.datを開いて，ファイル・タイプとファイルの中身を表示させます．

図6の1行目でファイル名を変数pathに代入します．

2行目のopen関数では，modeにbrを指定して，バイナリ・モードかつ，読み取り専用でファイルを開いて変数fにファイル・オブジェクトを代入します．

3行目でread関数で変数fのファイル・オブジェクトを文字列として返して，print関数で画面出力します．

5行目でファイルを閉じてファイルを解放します．

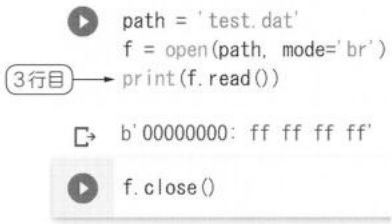


図6　ファイル読み出し関数4…バイナリ・モード＋read()

8-3　ファイル書き込み関数write関数/writelines関数

● ファイル書き込み関数の利用目的

Pythonプログラムでテキスト・ファイルやバイナリ・ファイルにデータを書き込む際には，write関数を使います．リストを書き込むwritelines関数もあります．これらは，I/Oストリームにデータを書き込むためのAPIを提供します．

I/Oストリームは，OSが管理するディスク，ファイル，デバイス，他のプログラム，メモリ配列などへ入出力するためプログラムが作成する論理インターフェースです．

このインターフェースを通じてプログラムは，OSを介してファイルへアクセスし，データを読み書きします．

● ファイル書き込み関数の書式

▶データを書き込む

```
write(data)
```

▶リストを書き込む

```
writelines(data)
```

● ファイル書き込み関数の解説

▶write関数

引数に文字列を指定した場合，それをI/Oストリームに書き込み，戻り値として書き込まれた文字数を返します．

バイナリ・データを指定すると同じようにI/Oストリームに書き込み，書き込まれたバイト数を返します．

▶writelines関数

引数に指定されたリストをI/Oストリームに書き込みます．行区切り文字（改行コード）は追加されません．書き込む各行の行末に，行区切り文字を含ませるのが一般的です．

● 実行例

ここではテキスト・ファイルのtest.txtとバイナリ・ファイルのtest.datを使います．ファイルの中身は前項と同じです．

▶文字列の書き込み

ここではtest.txtを開いて，新しい文字列を上書きします．

図7の1行目でファイル名を変数pathに代入し，2行目でopen関数でファイルを開いて変数fにファイル・オブジェクトを格納します．

ここではデフォルトのテキスト・モードでファイルが開かれます．modeにはwを指定します．

既にファイルが存在するので，ファイルの新規作成ではなく上書きになります．

3行目のwrite関数で変数fのファイル・オブジェクトが文字列としてNew lineを代入しています．

4行目には，write関数が返した文字数が表示されています．

5行目でファイルを閉じてファイルを解放します．

test.txtに上書きした情報が正しく書き込めているか確かめるため，6行目でopen関数でファイル

を開きます．

7行目で変数fのファイル・オブジェクトをread関数で読み出し，print関数で表示しています．9行目でファイルを閉じています．

このように既存のテキスト・ファイルに文字列を上書きできます．

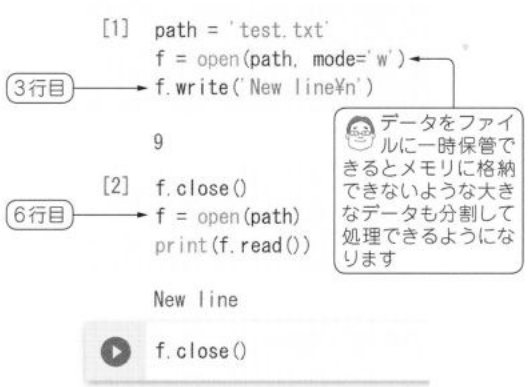


図7　ファイル読み書き関数1…文字列の書き込み

▶リストの書き込み

次にtest.txtにリストを上書きします．

図8の1行目でファイル名を変数pathに代入し，2行目でリストを変数lに代入します．

3行目で行の末尾に新しい文字列が追加されるようopen関数のmodeにaを指定します．戻り値のファイル・オブジェクトが変数fに格納されます．

4行目でwritelines関数の引数に変数lを指定して，変数fに書き込みます．

5行目でファイルを閉じてファイルを解放します．

7行目以降でtest.txtの内容を画面出力して，リストのオブジェクトが正常に書き込みできたことを確認します．

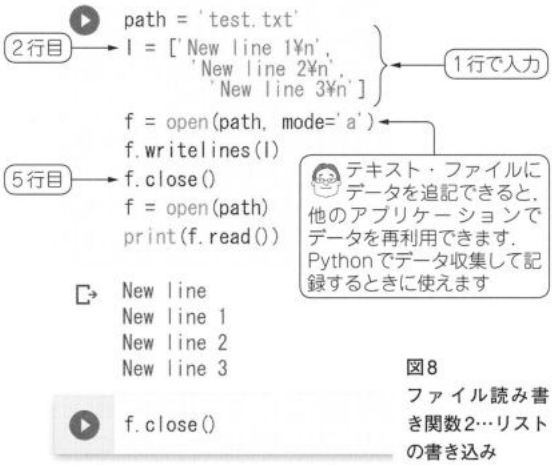


図8
ファイル読み書き関数2…リストの書き込み

▶バイナリ・ファイルの上書き

バイナリ・ファイルにバイナリ・データを上書きします．

図9の1行目でファイル名を変数pathに代入し，2行目のopen関数でmodeにファイル書き込みのwとバイナリ・モードのbを指定し，変数fにファイル・オブジェクトを格納します．

3行目でwrite関数にバイナリ・データを指定します．

4行目にバイト数が画面出力されています．

5行目でファイルを閉じてファイルを解放します．

7行目でtest.datの内容を画面出力して，バイナリ・データが正常に書き込みできたことを確認します．

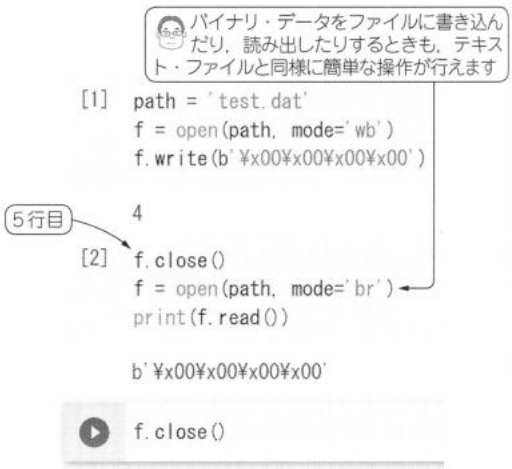


図9　ファイル読み書き関数3…バイナリ・ファイルの上書き

Pythonプログラムでファイル操作ができると，ファイルに記録されたデータを扱えるようになります．他のアプリケーションやプログラムが出力したファイルを入力データとしたり，開発したPythonプログラムが出力したデータを他のアプリケーションやプログラムで再利用したりできるようになります．

特にファイル書き込みは，Pythonプログラムで作ったグラフやデータの記録に関わる機能なので，利用価値の高い関数です．

ウェブで検索すると他の人が書いたPythonプログラムに，ファイル操作のソースコードを見つけられると思います．ソースコードを真似したり，それらを修正したりしてプログラムの機能を理解できると効率的に学習できます．

さとう・せい