2022 年度 【令和 4 年度】

第 3 学年　情報工学実験 II

奈良工業高等専門学校　情報工学科

目 次

第 I 章　実験報告書作成の手引 **iii**

第 1 章　オシロスコープに関する実験 (4 回):全教員 **1**

第 2 章　フリップフロップ回路に関する実験 (5 回):山口 (賢) **17**

第 3 章　オペアンプに関する実験 (4 回):山口 (賢) **29**

第 4 章　サイバーセキュリティに関する基礎実験 I (5 回):岡村 **39**

第 5 章　サイバーセキュリティに関する基礎実験 II (5 回):岡村 **41**

第 6 章　スクリプト言語を用いたテキストファイル処理に関する実験 (4 回):岩田 **43**

付録 **93**

付 録 A 実験機材・電子部品 **93**

付 録 B FGX-2005 クイックリファレンス **103**

付 録 C TTL-IC のピン配置 **105**

第6章　スクリプト言語を用いたテキストファイル処理に関する実験 (4 回):岩田

6.1　目的

本実験では，Python（パイソン）と呼ばれるスクリプト言語の文法や基本的な構造について学習し，Python を用いてさまざまなデータ整理や加工，テキストファイル処理およびツールの基本的な作成方法について習得することを目的とします.

6.2　班編成表

第 1 班	第 2 班	第 3 班	第 4 班	第 5 班	第 6 班
2	4	5	7	8	9
15	14	13	12	11	10
19	16	20	17	21	18
22	24	26	23	25	27
28	30	33	32	31	29
38	36	35	34	40	39
	41				

6.3　Python の実行方法

Python の実行方法には，以下に示す 2 通りの方法があります.

- Python の対話モードによる実行
- スクリプトファイルによる実行

6.3.1　対話モードによる実行

Windows 環境で Python を対話モードで実行するには,「コマンドプロンプト」を起動[1] し，図 6.1 に示すように python と入力します.

[1] スタートメニューから「すべてのプログラム」→「Windows システムツール」→「コマンドプロンプト」を選択

```
Microsoft Windows [Version 10.0.17763.529]
(c) 2018 Microsoft Corporation. All rights reserved.

Z:\>python
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52)
                                          [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 10 / 3                          # 計算式を入力（除算）
3.3333333333333335                  # 計算結果（Python 3.X 系の場合）
>>> exit ()                         # Python の終了

コマンドプロンプト>
```

図 6.1　Python の対話モードによる実行

Python の対話モードでは，Python のプロンプト「>>>」に式や Python のコマンドを直接入力します.（図 6.1 中の # 記号以降はコメントなので入力する必要はありません.）

対話モードを終了する場合には，Python のプロンプトにて，exit () と入力するか，もしくは「close ボタン」をクリックします.

6.3.2　スクリプトファイルによる実行

Windows 環境で Python をスクリプトファイルで実行するには，テキストエディタ（メモ帳やサクラエディタ，Tera Pad など）を使って Python の命令をテキストファイルに記述することにより，プログラム（スクリプト）として実行することができます．Python スクリプトのファイル拡張子は「.py」です.

Python スクリプトのサンプルをソースコード 6.1 に示します.

```
 1  #!/usr/local/bin/python3
 2
 3  #
 4  # Python のスクリプトファイル
 5  #
 6  #   四則演算（four_arithmetic.py）
 7  #
 8  print ("10␣+␣3␣=", 10 + 3)          # 和
 9  print ("10␣-␣3␣=", 10 - 3)          # 差
10  print ("10␣*␣3␣=", 10 * 3)          # 積
11  print ("10␣/␣3␣=", 10 / 3)          # 商
12  print ("10␣%␣3␣=", 10 % 3)          # 剰余
13  print ("10␣//␣3␣=", 10 // 3)        # 商の切り捨て
```

ソースコード 6.1　four_arithmetic.py

スクリプトファイルの 1 行目は,「インタプリタ宣言」で UNIX 系 OS で実行する場合には実行するコマンド（インタプリタ）を絶対パスで記述します.（Windows 環境で実行する場合や python コマンドの引数として実行する場合には不要です.）Python 2.X 系においては 2 行目に,「エンコード宣言」が記入されている場合がりますが，Python 3.X 系では省略してください．4 行目から 8 行目はコメントです．9 行目から 14 行目が Python の命令で，print 関数を使って計算結果を表示しています．Python では 1 行に 1 命令を記述します.

ソースコード 6.1 の Python スクリプトを実行するには，テキストエディタを使ってソースコード 6.1 の Python スクリプトを入力し，任意のフォルダ（Z: ドライブなど）に保存します.

次に，コマンドプロンプトを起動後，スクリプトファイルを保存したフォルダに移動し，以下のように入力します.

書式：python *script_file*

図 6.2 にソースコード 6.1 の Python スクリプトを Z: ドライブの Python フォルダに保存し，実行する場合の例を示します.

```
コマンドプロンプト> Z:                          # ドライブレターの変更
Z:\> mkdir Python                              # Python フォルダの作成
Z:\> cd Python                                 # Python フォルダへの移動
Z:\Python> dir                                 # Python フォルダ内容の表示

  ドライブ Z のボリューム ラベルは Network Drive です
  ボリューム シリアル番号は 2015-1225 です

  Z:\Python のディレクトリ

  2015/12/25  13:37    <DIR>          .
  2015/12/25  13:37    <DIR>          ..
  2015/12/25  20:48               329 four_arithmetic.py
             1 個のファイル                 329 バイト
             2 個のディレクトリ  123,456,789,000 バイトの空き領域

Z:\Python> python four_arithmetic.py      # Python スクリプトの実行
10 + 3 = 13
10 - 3 = 7
10 * 3 = 30
10 / 3 = 3.3333333333333335
10 % 3 = 1
10 // 3 = 3

Z:\Python>
```

図 6.2　Python スクリプト（four_arithmetic.py）の実行方法

ソースコード 6.1 のスクリプトが正常に実行されると，図 6.2 に示すように四則演算の結果が画

面に表示されます.（Python スクリプトを eclipse 環境で実行することもできます.）

スクリプトに間違いがあった場合には，図 6.3 に示すようなメッセージが表示され，スクリプトは終了します.

```
コマンドプロンプト> python four_arithmetic.py
  File "four_arithmetic.py", line 9
    print ("10 + 3 =": 10 + 3)                  # 和
                     ^
SyntaxError: invalid syntax

コマンドプロンプト>
```

図 6.3　Python スクリプトのエラーメッセージ例

図 6.3 の例では，four_arithmetic.py ファイルの 9 行目（「^」記号の箇所）に文法エラーがあることを示しています.

6.4　文法（基礎編）

6.4.1　コメント（注釈）

Python のコメントを記述するには，ハッシュ記号「#（半角）」を使います．ハッシュ記号「#は任意の場所に記述[2]することができ，以降行末までがコメントとして解釈されます.

複数行をコメントとして扱いたい場合には,「複数行文字列」とすることで，ブロックコメントのように扱うことができます．複数行文字列にするには，シングルクォーテーション「'（半角）」またはダブルクォーテーション「"（半角）」を 3 つ並べて記述します．ただし，複数行文字列として記述する場合は，書き始めるインデント（字下げ）位置に注意してください.

6.4.2　型とオブジェクト

Python では，数値や文字列などさまざまな「型」のデータを扱うことができます．これらすべての型を Python ではオブジェクトと呼んでいます.

標準で定義されているオブジェクトは「組み込み型」と呼ばれます．代表的な組込み型のオブジェクトを表 6.1 に示します.

[2]バックスラッシュによる継続行を除く

表 6.1　Python の組み込み型オブジェクト

オブジェクト			概要
数値	整数		メモリの制約は受けますが，精度の制限がありません.
		ブール値	真偽値の 0 (False) と 1 (True) のみを表現
	浮動小数点		decimal モジュールなどにより高精度を実現
	複素数		実部と虚部を持ち，虚部は 'j' や 'J' を使って表現
コンテナ	シーケンス	文字列	任意の文字列
		タプル	変更不可能なオブジェクトの集合
		リスト	変更可能なオブジェクトの集合
		bytes	バイナリー・データを表現します.
		バイト配列	
	集合	set	変更可能なオブジェクトの集合
		frozenset	変更不可能なオブジェクトの集合
マッピング	辞書		キーと値をペアにしたオブジェクトの集合

「コンテナ」とは，型の異なるオブジェクトの有限集合です.「シーケンス」は，順序を持ったオブジェクトの集合で,「集合」は，順序や重複する要素を持たないオブジェクトの集合です.

6.4.3　変数

Python では Java や C 言語などと違い，変数を使用する前に宣言をする必要はありません．変数には，半角英数字とアンダースコアを使うことができます．ただし，変数の先頭文字は必ず英字または，アンダースコアでなければなりません．また，大文字と小文字は区別されます．さらに，Java や C 言語と同様に，ローカル変数とグローバル変数は区別されますので有効範囲にも注意してください.

6.4.4　文字列

Python の文字列は，シングルクォーテーション「'」またはダブルクォーテーション「"」で文字列 *string ....* を囲みます．結果はどちらも同じ文字列になります.

書式：'*string ...*'
書式："*string ...*"

文字列は，変数に代入することができます.

6.4.5　タプル

タプルは，複数のデータを一列に並べたもので，型の異なる複数のデータをひとつにまとめて取り扱うことができます.

タプルは，角括弧「( )」で囲んだ中にデータ *value_item* をカンマ「,」で区切って記述します. タプルに格納されたデータ *value_item* を「要素」といいます.

書式：(*value_item_1*，<*value_item_2*，･･･，*value_item_n*>)

要素数が 1 個だけの場合には，(*value_item*,) と記述します．また，*value_item* にタプルオブジェクトを記述することで多次元配列のタプルオブジェクトを作成することができます.
上記以外にも Python の組み込み関数である `tuple` 関数を使用することもできます.

書式：`tuple ("`*value_item*`")`
書式：`tuple (`*sequence_value_item*`)`

`tuple` 関数では，*value_item* に文字列を指定すると，各文字を要素とするタプルを作成します. また，*sequence_value* にリストオブジェクトを指定するとタプルオブジェクトに変換します.
タプルも文字列と同様，変数に代入することができますが，タプルオブジェクトは変更することができません.

6.4.6　リスト

リストは，タプルと同じく，型の異なる複数のデータをひとつにまとめて取り扱うことができます.

リストは，角括弧「[ ]」で囲んだ中にデータ *value_item* をカンマ「,」で区切って記述します.

書式：[*value_item_1*，<*value_item_2*，･･･，*value_item_n*>]

要素数が 1 個だけの場合には，[*value_item*,] と記述します．また，*value_item* にリストオブジェクトを記述することで多次元配列のリストオブジェクトを作成することができます.
リストも文字列やタプルと同様，変数に代入することができます.
上記以外にも Python の組み込み関数である `list` 関数を使用することもできます.

書式：`list ("`*value_item*`")`
書式：`list (`*sequence_value*`)`

list 関数では，*value_item* に文字列を指定すると，各文字を要素とするリストを作成します．また，*sequence_value* にタプルオブジェクトを指定するとリストオブジェクトに変換します．リストの操作にはさまざまなメソッドが用意されています．

append メソッド　引数で指定された *value_item* をリスト *list_object* の末尾に追加します．

書式：*list_object*.append (*value_item*)

extend メソッド　引数で指定された複数の *value_items ...* をリスト *list_object* の末尾に追加します．

書式：*list_object*.extend ([*value_item_1*<, *value_item_2*, ··· *value_item_n*>])

また，加算演算子「+」や乗算演算子「*」を使用することもできます．

書式：*list_object* + *list_object*
書式：*list_object* * *n*

list_object に整数値 *n* を指定すると *list_object* を *n* 回繰り返すことができます．

insert メソッド　指定した位置に要素を挿入します．

書式：*list_object*.insert (*index*, *value_item*)

insert メソッドは，*index* で指定されたインデックス位置の前に要素 *value_item* を挿入します．

pop メソッド　リスト *list_object* から指定されたインデックスの要素を取り除きます．

書式：*list_object*.pop (<*index*>)

index を省略した場合は，末尾の要素を取り除きます．

remove メソッド　リスト *list_object* の中から値 *value_item* をもつ最初の要素を削除します．

書式：*list_object*.remove (*value_item*)

sort メソッド　リスト *list_object* の要素を昇順（デフォルト）にソートします.

書式：*list_object*.sort （<reverse=True>）

sort メソッドは返り値を取らず, 元の要素と置き換えます. また, sort メソッドの引数に reverse=True を指定すると降順にソートすることができます.

reverse メソッド　リスト *list_object* の要素を降順にソートします.

書式：*list_object*.reverse （）

リストのサンプルをソースコード 6.2 に示します.

```
1  #!/usr/local/bin/python3
2
3
4  print ("◆␣リストの操作")
5  lname = []                          # 空のリストオブジェクトの作成
6  print ("【1】：", lname, "\n")
7
8  print ("◆␣「append␣メソッド」によるリストの追加")
9  lname.append ("2016/01/15") # 空のリストオブジェクトにデータ 2016/01/15 を追加
10 lname.append ("2016/01/22") # さらにデータ 2016/01/22 を追加
11 print ("【2】：", lname, "\n")
12
13 print ("◆␣「extend␣メソッド」によるリストの追加")
14 lname.extend (["2016/01/08", "2016/01/29"]) # リストデータの追加
15 print ("【3】：", lname, "\n")
16
17 print ("◆␣「sort␣と␣reverse␣メソッド」によるリストの並び替え")
18 lname.sort ()                       # リストデータのソート (1)
19 print ("【4】：", lname)
20 lname.reverse ()                    # リストデータのソート (2)
21 print ("【5】：", lname, "\n")
22
23 print ("◆␣「index␣メソッド」によるインデックスの取得")
24 record = lname.index ("2016/01/15") # 要素の格納インデックスの取得
25 print ("【検索結果】", record, "\n")
26
27 print ("◆␣「pop␣メソッド」による値の取得")
28 lname.pop ()                        # 末尾の要素の取り出し
29 print ("【6】：", lname)
30 lname.pop (1)                       # 1 番目のインデックス要素の取り出し
31 print ("【7】：", lname, "\n")
```

ソースコード 6.2　list_sample.py

```
コマンドプロンプト> python list_sample.py
◆ リストの操作
【1】: []

◆ 「append メソッド」によるリストの追加
【2】: ['2016/01/15', '2016/01/22']

◆ 「extend メソッド」によるリストの追加
【3】: ['2016/01/15', '2016/01/22', '2016/01/08', '2016/01/29']

◆ 「sort と reverse メソッド」によるリストの並び替え
【4】: ['2016/01/08', '2016/01/15', '2016/01/22', '2016/01/29']
【5】: ['2016/01/29', '2016/01/22', '2016/01/15', '2016/01/08']

◆ 「index メソッド」によるインデックスの取得
【検索結果】 2

◆ 「pop メソッド」による値の取得
【6】: ['2016/01/29', '2016/01/22', '2016/01/15']
【7】: ['2016/01/29', '2016/01/15']

コマンドプロンプト>
```

図 6.4　list_sample スクリプトの実行結果

文字列やリスト，タプルなどのシーケンス型オブジェクトには，以下に示すように格納された先頭の要素からインデックス（添字）が割り当てられます．Python では，正のインデックスと負のインデックスの 2 種類があります．

正インデックス	0	1	2	3
文字列オブジェクト	N	I	o	T
リストオブジェクト	National	Institute	of	Technology.
負インデックス	-4	-3	-2	-1

index メソッド　シーケンス型オブジェクト *object* から引数で指定された *value_item* が格納された要素（インデックス）を取得します．

書式：*object*.index（*value_item*）

文字列やリスト，タプルなどのシーケンス型オブジェクトに格納された要素はインデックスを使って参照することができます．

また，リストオブジェクトに格納された要素はインデックスを使って変更することができます．（ソースコード 6.3 参照）

```
1  #!/usr/local/bin/python3
2
3
4  lname = ["National", "Institute", "of", "Technology."]  # リストオブジェクト
5
6  print ("■␣リストオブジェクト␣lname␣の参照")
7  print ("lname:␣", lname, "\n")
8
9  print ("■␣リストオブジェクト␣lname␣のインデックスを使った参照")
10 print ("lname[0]:", lname[0])           # ◆ 正インデックスを使った参照
11 print ("lname[1]:", lname[1])           # ◆ 正インデックスを使った参照
12 print ("lname[-2]:", lname[-2])         # ◆ 負インデックスを使った参照
13 print ("lname[-1]:", lname[-1], "\n")   # ◆ 負インデックスを使った参照
14
15 print ("■␣リストオブジェクト␣lname␣のインデックスを使った要素の変更")
16 lname[0] = "Depertment"
17 lname[1] = "of"
18 lname[2] = "Information"
19 lname[3] = "Engeneering"
20 print ("lname:␣", lname)
21 print ("lname[0]:", lname[0])
22 print ("lname[1]:", lname[1])
23 print ("lname[2]:", lname[2])
24 print ("lname[3]:", lname[3], "\n")
```

ソースコード 6.3　index_sample.py

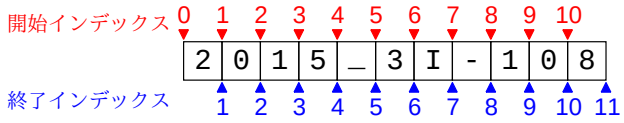
```
コマンドプロンプト> python index_sample.py
■ リストオブジェクト lname の参照
lname:  ['National', 'Institute', 'of', 'Technology.']

■ リストオブジェクト lname のインデックスを使った参照
lname[0]: National
lname[1]: Institute
lname[-2]: of
lname[-1]: Technology.

■ リストオブジェクト lname のインデックスを使った要素の変更
lname:  ['Depertment', 'of', 'Information', 'Engeneering']
lname[0]: Depertment
lname[1]: of
lname[2]: Information
lname[3]: Engeneering

コマンドプロンプト>
```

図 6.5　index_sample スクリプトの実行結果

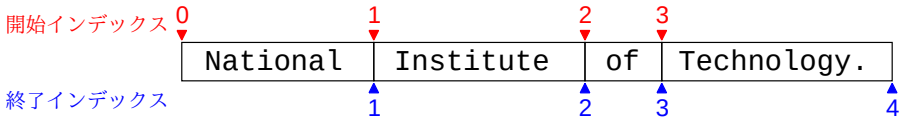
6.4.7　スライス

Python ではシーケンス型オブジェクトのインデックスの範囲を指定して文字列やリストの要素を切り出すことができます．これを「スライス」と言います．

インデックスの範囲は，開始インデックスと終了インデックスで指定します．シーケンス型オブジェクトの開始インデックスと終了インデックスを図 6.6 に示します．



(a) 文字列オブジェクトの場合



(b) リストオブジェクトの場合

図 6.6　シーケンス型オブジェクトにおける開始インデックスと終了インデックス

スライスは，要素を切り出す対象のシーケンスオブジェクト（*sequence_object*）と角括弧「[ ]」の中に，開始インデックス（*start_index*），コロン「:」，終了インデックス（*end_index*）の順番で記述します．

書式：*sequence_object*[<*start_index*>:<*end_index*>]

start_index や *end_index* は，省略することができます．*start_index* が省略された場合には 先頭インデックスの「0」が指定されたことになります．また，*end_index* が省略された場合には，その文字列やリストの「末尾インデックス」が指定されたことになります．(ソースコード 6.4 参照)

```
1  #!/usr/local/bin/python3
2
3
4  #
5  # インデックスを使った文字列やリストのスライス
6  #
7  print ("■␣文字列オブジェクトのスライス")
8
9  string = "2015_3I-108"       # 文字列オブジェクト
10
11 print (string[5:7])          # 開始インデックスと終了インデックスを指定
12 print (string[:4])           # 開始インデックスを省略
13 print (string[8:], "\n")     # 終了インデックスを省略
14
15 print ("■␣リストオブジェクトのスライス")
16
```

```
17  l_name = ["National", "Institute", "of", "Technology."] # リストオブジェクト
18
19  print (l_name[1:3])         # 開始インデックスと終了インデックスを指定
20  print (l_name[:2])          # 開始インデックスを省略
21  print (l_name[2:])          # 終了インデックスを省略
```

ソースコード 6.4　slice_sample.py

```
コマンドプロンプト> python slice_sample.py
■ 文字列オブジェクトのスライス
3I
2015
108

■ リストオブジェクトのスライス
['Institute', 'of']
['National', 'Institute']
['of', 'Technology.']

コマンドプロンプト>
```

図 6.7　slice_sample スクリプトの実行結果

6.4.8　集合型

Python の組み込み型の一つである集合は，重複する要素を持たず，順序付けされていない要素の集まりを扱うためのオブジェクトです．集合型には，set と frozenset の 2 種類があります．

書式：set ([*value_item_1*<, *value_item_2*, ···, *value_item_n*>])
書式：frozenset ([*value_item_1*<, *value_item_2*, ···, *value_item_n*>])

set または frozenset は *value_item* から要素を取り込んだ，新しいオブジェクトを返します．

set オブジェクトは，変更可能で add や remove メソッドなどを使って要素を変更することができます．frozenset オブジェクトは，変更することができませんが，ハッシュ化可能なため辞書のキーや他の集合の要素として使用することができます．

集合オブジェクトは，以下に示すような演算をサポートしています．

in, not in 演算子　in 演算子は，要素（*value_item*）が集合オブジェクト（*object*）のメンバーに含まれているか判定しブール値（「True」or「False」）で返します．

not in 演算子は，要素（*value_item*）が集合オブジェクト（*object*）のメンバーに含まれていないことを判定しブール値（「True」or「False」）で返します．

書式：*value_item* in *object*
書式：*value_item* not in *object*

in, not in 演算子のサンプルをソースコード 6.5 に示します.

```
1  #!/usr/local/bin/python3
2
3
4  print ("■␣set：重複要素を取り除く［順序も持たない］")
5  school_name = set ('Nara␣College')          # 単一の文字列
6  depertment = set ([                         # 複数の文字列
7      'Liberal', 'Studies',           'Mechanical', 'Engineering',
8      'Electrical', 'Engineering',    'Control', 'Engineering',
9      'Information', 'Engineering',   'Chemical', 'Engineering'
10 ])
11
12 print ("Check␣A:␣", school_name)             # 単一の文字列の場合
13 print ("Check␣B:␣", depertment, "\n")        # 複数の文字列の場合
14
15 print ("■␣in,␣not␣in␣演算子")
16 print ("Check␣C:", 'Information' in depertment)
17 print ("Check␣D:", 'Architecture' in depertment)
18 print ("Check␣E:", 'Architecture' not in depertment, "\n")
```

ソースコード 6.5　set_in_sample.py

サンプルをソースコード 6.5 の実行結果を図 6.8 に示します.（紙面の都合上，改行しています.）

```
コマンドプロンプト> python set_in_sample.py
■ 重複要素を取り除く［順序も持たない］
Check A:  {' ', 'l', 'g', 'o', 'e', 'C', 'N', 'a', 'r'}
Check B:  {'Information', 'Liberal', 'Mechanical', 'Control', 'Chemical',
           'Engineering', 'Studies', 'Electrical'}

■ in, not in 演算子
Check C: True
Check D: False
Check E: True

コマンドプロンプト>
```

図 6.8　set_in_sample スクリプトの実行結果

非交和 isdisjoint メソッド　集合がもう片方のオブジェクトと共通の要素を持たない（非交和）とき，True を返します.

書式：*object*.isdisjoint (*other_object*)

部分集合 issubset, issuperset メソッド　集合オブジェクトの全ての要素が，もう片方の集合オブジェクトに含まれるかを判定しブール値（「True」or「False」）で返します.

書式：*object*.issubset (*other_object*)
書式：*object* < *other_object*
書式：*object* <= *other_object*
書式：*object*.issuperset (*other_object*)
書式：*object* > *other_object*
書式：*object* >= *other_object*

issubset メソッドは，集合オブジェクト *object* の（全ての）要素が，もう片方の集合オブジェクト *other_object* に含まれるか判定します.
issuperset メソッドは，もう片方の集合オブジェクト *other_object* の（全ての）要素が，集合オブジェクト *object* に含まれるか判定します.

部分集合を判定する場合には，演算子（<，<=，>，>=）を使用することもできます.

論理和 union メソッド　集合オブジェクトともう片方の集合オブジェクト全ての要素からなる新しい集合を返します.

書式：*object*.union (*other_object*<, *other_object_1*, ···, *other_object_n*>)
書式：*object* | *other_object*< | ···>

論理差 difference メソッド　集合オブジェクトに含まれて，かつ，もう片方の集合オブジェクトに含まれない要素を持つ新しい集合を返します.

書式：*object*.difference (*other_object*<, *other_object_1*, ···, *other_object_n*>)
書式：*object* - *other_object*< - ···>

論理積 intersection メソッド　集合オブジェクトともう片方の集合オブジェクトに共通する要素を持つ新しい集合を返します.

書式：*object*.intersection (*other_object*<, *other_object_1*, ···, *other_object_n*>)
書式：*object* & *other_object*< & ···>

排他的論理和 symmetric_difference メソッド　集合オブジェクトともう片方の集合オブジェクトのいずれか一方に含まれる要素を持つ新しい集合を返します.

書式：*object*.symmetric_difference (*other_object*<, *other_object_1*, ···, *other_object_n*>)
書式：*object* ^ *other_object*< ^ ···>

集合オブジェクトでサポートする各種演算子のサンプルをソースコード 6.6 に示します.

```
#!/usr/local/bin/python3


print ("■␣非交和␣isdisjoint␣演算子")
depertment = set ([                 # 複数の文字列
    'Liberal', 'Studies',       'Mechanical', 'Engineering',
    'Electrical', 'Engineering',    'Control', 'Engineering',
    'Information', 'Engineering',   'Chemical', 'Engineering'
])
depertment_l = set (['Liberal', 'Studies'])
depertment_m = set (['Mechanical', 'Engineering'])
depertment_e = set (['Electrical', 'Engineering'])
depertment_s = set (['Control', 'Engineering'])
depertment_i = set (['Information', 'Engineering'])
depertment_c = set (['Chemical', 'Engineering'])

print ("Check␣F␣(dep_i␣?␣dep_m)：", depertment_i.isdisjoint (depertment_m))
print ("Check␣G␣(dep_i␣?␣dep_l)：", depertment_i.isdisjoint (depertment_l))
print ("")

print ("■␣部分集合␣issubset,␣issuperset␣メソッド")
depertment_x = set (['Architecture', 'Engineering'])

print ("Check␣H␣(dep_i␣?␣dep)：", depertment_i.issubset (depertment))
print ("Check␣I␣(dep_x␣?␣dep)：", depertment_x.issuperset (depertment))
print ("")

print ("■␣論理和␣union␣メソッド")
union_ei = depertment_e.union (depertment_i)
union_ms = depertment_m | depertment_s

print ("Check␣J␣(dep_e␣?␣dep_i)：", union_ei)
print ("Check␣K␣(dep_m␣?␣dep_s)：", union_ms)
print ("")

print ("■␣論理差␣difference␣メソッド")
difference_ei = depertment_e.difference (depertment_i)
difference_ms = depertment_m - depertment_s

print ("Check␣L␣(dep_e␣?␣dep_i)：", difference_ei)
print ("Check␣M␣(dep_m␣?␣dep_s)：", difference_ms)
print ("")
```

```
43
44 print ("■␣論理積␣intersection␣メソッド")
45 intersection_ei = depertment_e.intersection (depertment_i)
46 intersection_ms = depertment_m & depertment_s
47
48 print ("Check␣N␣(dep_e␣?␣dep_i)：", intersection_ei)
49 print ("Check␣O␣(dep_m␣?␣dep_s)：", intersection_ms)
50 print ("")
51
52 print ("■␣排他的論理和␣symmetric_difference␣メソッド")
53 symmetric_difference_ei = depertment_e.symmetric_difference (depertment_i)
54 symmetric_difference_ms = depertment_m ^ depertment_s
55
56 print ("Check␣P␣(dep_e␣?␣dep_i)：", symmetric_difference_ei)
57 print ("Check␣Q␣(dep_m␣?␣dep_s)：", symmetric_difference_ms)
58 print ("")
```

ソースコード 6.6　set_operator_sample.py

サンプルをソースコード 6.6 の実行結果を図 6.9 に示します.

```
コマンドプロンプト> python set_operator_sample.py
■ 非交和 isdisjoint 演算子
Check F (dep_i ? dep_m)： False
Check G (dep_i ? dep_l)： True

■ 部分集合 issubset, issuperset メソッド
Check H (dep_i ? dep)： True
Check I (dep_x ? dep)： False

■ 論理和 union メソッド
Check J (dep_e ? dep_i)： {'Engineering', 'Electrical', 'Information'}
Check K (dep_m ? dep_s)： {'Mechanical', 'Engineering', 'Control'}

■ 論理差 difference メソッド
Check L (dep_e ? dep_i)： {'Electrical'}
Check M (dep_m ? dep_s)： {'Mechanical'}

■ 論理積 intersection メソッド
Check N (dep_e ? dep_i)： {'Engineering'}
Check O (dep_m ? dep_s)： {'Engineering'}

■ 排他的論理和 symmetric_difference メソッド
Check P (dep_e ? dep_i)： {'Electrical', 'Information'}
Check Q (dep_m ? dep_s)： {'Mechanical', 'Control'}

コマンドプロンプト>
```

図 6.9　set_operator_sample スクリプトの実行結果

6.4.9　辞書（ハッシュ，連想配列）

Python の組み込み型の一つである辞書は,「キー」と「値」をペアにしたオブジェクトの集合で,リストのような要素の順番はありません.

辞書は，中括弧「{ }」の間に「キー（*key*）：値（*value_item*）」をペアとして，カンマ「,」で区切って記述します.

書式：{*key_1*:*value_item_1*，*key_2*:*value_item_2*，･･･，*key_n*:*value_item_n*}

上記以外にも Python の組み込み関数である `dict` 関数を使用することもできます.

書式：`dict ([(`*key_1*，*value_item_1*`), (`*key_2*，*value_item_2*`), ･･･, (`*key_n*，*value_item_n*`)])`
書式：`dict (`*key_1*`=`*value_item_1*，*key_2*`=`*value_item_2*，･･･，*key_n*`=`*value_item_n*`)`

key は必ず一意であり，同じ値のキーが複数存在することはありません．また，*key* には，数値や文字列およびタプルなどを使用することができます.

値 *value_item* については，全てのオブジェクトを指定することができます.

辞書の操作には以下に示すようなさまざまなメソッドが用意されています.

get メソッド　辞書オブジェクト *dict_object* の *key* に対応する *value* を取得します.

書式：*dict_object*`.get (`*key*`<`，*return_value*`>)`

return_value は，*key* がなかったときに get メソッドが返す値です．省略すると `None` を記述したことになります.

keys メソッド　辞書オブジェクト *dict_object* のキーの一覧を取得します.

書式：*dict_object*`.keys ()`

values メソッド　辞書オブジェクト *dict_object* の要素の一覧を取得します.

書式：*dict_object*`.values ()`

items メソッド　辞書オブジェクト *dict_object* の要素のキーと要素の一覧を取得します.

書式：*dict_object*.items ()

update メソッド　*dict_target_object* と引数で指定された *dict_source_object* を結合します.

書式：*dict_target_object*.update (*source_object*)

ソースコード 6.7 に辞書のサンプルを示します.

```
1  #!/usr/local/bin/python3
2
3
4  general= {           # 一般教科の辞書オブジェクト
5       "3I003":"国語␣III",            "3I031":"英語␣III",
6  }
7
8  specialized= {        # 専門教科の辞書オブジェクト
9      "3I109":"プログラミング",     "3I115":"情報数学␣II",
10 }
11
12 syllabus = {}        # 空の辞書オブジェクト
13
14 print ("◆␣「キー」による値の取得")
15 print ("一般科目␣[3I031]：", general["3I031"], "\n")
16
17 print ("◆␣「get␣メソッド」による値の取得")
18 print ("専門科目␣[3I136]：", specialized.get ("3I136"), "\n")     # 未登録のキーを指定
19
20 specialized["3I109"] = "プログラミング␣II"                        # 要素の変更
21 specialized["3I136"] = "情報工学実験␣II"                          # 要素の追加
22
23 print ("◆␣「keys␣メソッド」によるキーの一覧を取得")
24 print ("一般科目：", general.keys ())
25 print ("専門科目：", specialized.keys (), "\n")
26
27 print ("◆␣「values␣メソッド」による値の一覧を取得")
28 print ("専門科目：", specialized.values (), "\n")
29
30 print ("◆␣「items␣メソッド」によるキーと値の一覧を取得")
31 print ("一般科目：", general.items (), "\n")
32
33 print ("◆␣「update␣メソッド」による辞書の結合")
34 print ("【結合前】syllabus：", syllabus.keys ())
35 syllabus.update (general)                                        # 辞書の結合
36 syllabus.update (specialized)                                    # 辞書の結合
37 print ("【結合後】syllabus：", syllabus.keys (), "\n")
38
```

```
39  print ("◆␣「del」による辞書オブジェクトや辞書オブジェクト要素の削除")
40  del specialized["3I115"]                          # 辞書オブジェクト「要素」の削除
41  del general                                       # 辞書オブジェクトの削除
42  print ("専門科目：", specialized.keys (), "\n")
```

ソースコード 6.7　dictionary_sample.py

```
コマンドプロンプト> python dictionary_sample.py
◆ 「キー」による値の取得
一般科目 [3I031]： 英語 III

◆ 「get メソッド」による値の取得
専門科目 [3I136]： None

◆ 「keys メソッド」によるキーの一覧を取得
一般科目： dict_keys(['3I031', '3I003'])
専門科目： dict_keys(['3I109', '3I115', '3I136'])

◆ 「values メソッド」による値の一覧を取得
専門科目： dict_values([' プログラミング II', ' 情報数学 II', ' 情報工学実験 II'])

◆ 「items メソッド」によるキーと値の一覧を取得
一般科目： dict_items([('3I031', ' 英語 III'), ('3I003', ' 国語 III')])

◆ 「update メソッド」による辞書の結合
【結合前】syllabus： dict_keys([])
【結合後】syllabus： dict_keys(['3I109', '3I115', '3I031', '3I136', '3I003'])

◆ 「del」による辞書オブジェクトや辞書オブジェクト要素の削除
専門科目： dict_keys(['3I109', '3I136'])

コマンドプロンプト>
```

図 6.10　dictionary_sample スクリプトの実行結果

6.4.10　ブロック（複合文）

Python では，Java や C 言語とは異なり，単一の文とみなされるブロックは「{ … }」（中カッコ）を用いるのではなく，インデント（字下げ）を用います.

インデントは，一レベルインデントするごとに半角スペース 4 つが推奨されています.

Python 3.X 系では，タブと半角スペースが混在しているインデントはエラーになります.

6.4.11　条件分岐 if 文

Python では条件を判断して分岐処理を行うには `if` 文を用います．Java や C 言語で使われる `switch` 〜 `case` 文はありません.

書式：<*initialization*>
```
    if first_expression:
        first_statement
    <elif expression_1:
        statement_1>
          :
    <elif expression_n:
        statement_n>
    <else:
        else_statement>
```

`if` 文は，条件 *first_expression* が「真」（True）なら字下げされた *first_statement* の範囲（ブロック）を実行します．*first_expression* が「偽」（False）なら，次の `elif` の条件 *expression_1* が評価されます．どの条件も成立しない場合には，`else:` の字下げされた *else_statement* の範囲（ブロック）を実行します．また，Python では条件 *expression* において比較演算子を連続して記述をすることができます.（ソースコード 6.8 参照）

`elif` と `else` は省略可能で，`elif` は必要に応じていくつでも記述することができます.

6.4.12　繰り返し処理

Python における繰り返し処理には，`for` 文と `while` 文の 2 通りがあります．Java や C 言語で使われる `do` 〜 `while` 文はありません.

for 文

`for` 文は，与えられたシーケンス型オブジェクト *container* すべての要素に対して，要素がなくなるまで繰り返しブロック内の処理を行います.

書式：`for` *v_name* `in` *container*:
```
        statement
```

`for` 文は，変数 *v_name* にコンテナ *container* の要素を先頭から一つずつ入れながら， 字下げされた範囲（ブロック）の文 *statement* をすべての要素に対して行います.

while 文

`while` 文は条件が「真」（True）の間，繰り返しブロック内の処理を行います.

書式：<*initialization*>
`while` *expression*:
statement
<*step*>

initialization はループを開始する直前に実行する初期化処理で，必要に応じて記述します．
その後，条件式 *expression* が「真」（Ture）の間，字下げされた範囲（ブロック）の文 *statement* やカウンタ処理 *step* を実行します．

Python では，無限ループを行う場合に `while` 文を使用します．

break 文と continue 文

Python でも Java や C 言語と同様 `for` や `while` の繰り返し処理において `break` 文と `continue` 文を使用することができます．
繰り返し文の中で，`break` 文が実行されると，最も内側の `for` または `while` ループを中断します．また，繰り返し文の中で，`continue` 文が実行されると，ループのそれ以降の処理を行わず，ループを次の反復処理に飛ばします．

else 節

Python では，`for` や `while` の繰り返し処理においても `else` 節を使用することができます．

書式：`else:`
statement

`else` 節は，`for` 文や `while` 文の処理中で `break` 文が使用されなかった時に,`else` 節のブロックを実行します．（ソースコード 6.8 参照）

```
1  #!/usr/local/bin/python3
2
3
4  print ("■␣for␣文で␣break␣しない場合")
5  for m in range (10):                    # 10 回（0 〜 9 まで）の繰り返し
6      print ("Stage␣A:", m)
7
8  else:
9      print ("Stage␣B:␣for␣...␣else␣statement")
10     print ("Stage␣B:␣Loop␣Complete!")
11
12 print ("Stage␣C:␣done.\n")
```

```
13
14
15 print ("■␣for␣文で␣break␣する場合")
16 for n in range (10):
17     print ("Stage␣D:", n)
18
19     #
20     # 条件式には数学的「 1 < n < 5 」な記述をすることができます.
21     #
22     if ((1 < n < 5) and ((n % 2) == 0)):    # 偶数なら break
23         print ("Stage␣E:␣Break␣.....", n)
24         break
25
26 else:
27     print ("Stage␣F:␣for␣...␣else␣statement")
28     print ("Stage␣F:␣Loop␣Complete!")
29
30 print ("Stage␣G:␣done.\n")
```

ソースコード 6.8　else_sample.py

```
コマンドプロンプト> python else_sample.py
■ for 文で break しない場合
Stage A: 0
Stage A: 1
Stage A: 2
Stage A: 3
Stage A: 4
Stage A: 5
Stage A: 6
Stage A: 7
Stage A: 8
Stage A: 9
Stage B: for ... else statement
Stage B: Loop Complete!
Stage C: done.

■ for 文で break する場合
Stage D: 0
Stage D: 1
Stage D: 2
Stage E: Break ..... 2
Stage G: done.

コマンドプロンプト>
```

図 6.11　else_sample スクリプトの実行結果

6.5　標準入力操作

キーボードからデータなどの入力を受け取るには，input 関数を使用します.

input 関数　キーボードからの入力を受け取り，その入力を Python の式として解釈し，その評価結果を返します.

書式：input（<*prompt*>）

prompt は，キーボードからの入力を受け取る時に標準出力に表示されるプロンプトです．*prompt* は省略することもできます．受け取った入力は評価を行い文字列に変換して返します.

ソースコード 6.9 に，input 関数による標準入力の受け取り例を示します.

```
1  #!/usr/local/bin/python3
2
3
4  sx = input ("整数␣x：␣")        # 文字列として受け取る
5  sy = input ("整数␣y：␣")        # 文字列として受け取る
6
7  #
8  # 入力データのチェック
9  #
10 #   整数でなかったらエラーメッセージを表示して終了
11 #
12 if ((sx.isdigit () == False) or (sy.isdigit () == False)):
13     print ("整数を入力してください．")
14     exit (1)
15
16 #
17 # 文字列を数値（10 進数）に変換
18 #
19 ix = int (sx)
20 iy = int (sy)
21
22 print ("【数値】x␣+␣y␣=␣", ix + iy)
23 print ("【数値】x␣-␣y␣=␣", ix - iy)
24 print ("【数値】x␣*␣y␣=␣", ix * iy)
25 print ("【数値】x␣/␣y␣=␣", ix / iy)
26 print ("【数値】x␣%␣y␣=␣", ix % iy)
27 print ("【数値】x␣//␣y␣=␣", ix // iy, "\n")
28
29 print ("【文字列】x␣+␣y␣=␣", sx + sy)     # 文字の連結
```

ソースコード 6.9　input_sample.py

```
コマンドプロンプト> python input_sample.py
整数 x： 20
整数 y： 15
【数値】x + y =  35
【数値】x - y =  5
【数値】x * y =  300
【数値】x / y =  1.3333333333333333
【数値】x % y =  5
【数値】x // y =  1

【文字列】x + y =  2015

コマンドプロンプト>
```

図 6.12　input_sample スクリプトの実行結果

6.6　ファイル操作

データが記録されたファイルを読み込んだり，処理結果やエラーログなどのファイルを扱うには，open 関数と close 関数をペアで使用します．

書式：<*file_object* => open (*filename*<, *mode*><, encoding=*file_encode*>)
　　statement
書式：*file_object*.close ()

file_object はファイルオブジェクトで，このファイルオブジェクトを通してファイルの読み書きを行ないます．

open 関数は引数に対象となるファイル *filename* と必要に応じて，動作モード *mode* やファイルの文字コード *file_encode* を指定します．動作モードを省略した場合には,「テキストモードの読み込み」となります．

open 関数の動作モードを表 6.2 に示します．

表 6.2　open 関数の動作モード

動作モード記号	意味
'r'	読み込み用（デフォルト）
't'	テキストモード（デフォルト）
'w'	書き込み用（ファイルが存在しない場合には新規作成，ファイルが存在している場合は上書き）
'+'	更新用（読み込み/書き込み）
'a'	追加書き込み用（ファイルが存在している場合は末尾に追記）
'b'	バイナリモード

ファイルの内容を読み出すには，以下に示すような様々なメソッドがあります.

read メソッド　ファイルオブジェクトから読み込んで一つの文字列オブジェクトを返します.

書式：*file_object*.read (<*buffer_size*>)

buffer_size は読み込み時のバッファサイズで，*buffer_size* が指定されると，*file_object* から *buffer_size* バイト読み込んで一つの文字列オブジェクトとして返します．*buffer_size* が省略された場合には，EOF（End of File）まで読み込みます.

readline メソッド　ファイルオブジェクトから改行または EOF までを読み込んで一つの文字列オブジェクトを返します.

書式：*file_object*.readline ()

readlines メソッド　ファイルオブジェクトから全てを一度に読み込んで，それらを行単位で区切り，リストとして返します.

書式：*file_object*.readlines ()

write メソッド　ファイルオブジェクトにデータを書き込みます.

書式：*file_object*.write (*string*)

write メソッドは，引数で指定した *string* を *file_object* に書き込みます．*string* は文字列型でなければなりません.

with 文

従来のファイル操作は，try ··· except ··· finary 節などと併用してファイルのオープンとクローズを行っていましたが，Python 3.X 系では with 文を使用することができます．

with 文は，ファイルのオープンに対応するクローズ処理を自動的に行ないます．そのためクローズ処理に関する記述を省略することができます．

書式：with *action* as *file_object*:
　　statement

with 文でファイル操作を行うには，*action* に open 関数を記述します．ファイルの読み書きには，*file_object* を通して行ないます．

action の評価が「真」（True）の間，字下げされた *statement* の範囲（ブロック）を実行し，*action* の評価が「偽」（False）になると，ブロックを抜けファイルのクローズ処理を自動的に行います．

ソースコード 6.10 に，指導書 77 のソースコード 6.14 の実行結果をファイルに保存した気温データを読み込み平均気温と最高気温の日時をファイルに書き出すサンプルスクリプトを示します．

```
 1  #!/usr/local/bin/python3
 2
 3
 4  import os.path                              # os.path モジュールをインポート
 5
 6  #
 7  # 変数の初期化
 8  #
 9  n = 0                                      # 日数のカウンタ
10  total = 0.0                                # 気温の合計
11  max = 0.0                                  # 最高気温
12  average = 0.0                              # 平均気温
13
14  #
15  # エラー処理
16  #
17  #   ファイル読み込み失敗時
18  #   読み込むファイルがなかったらエラーメッセージを表示して終了
19  #
20  if (os.path.isfile ("temperature.dat") == False):
21      print ("temperature.dat:␣No␣such␣file␣or␣directory.")
22      exit (1)
23
24  #
25  # ファイルの読み込み
26  #
27  with open ("temperature.dat", 'r') as r_file:       # ファイルを読み込みモードでオープン
28
29      for line_data in r_file:                # ファイルの終わりまで読み込みを繰り返す
```

```
30          daily_data = line_data.split ()     # リストに変換
31
32          if daily_data[1] == "気温［℃］":     # ヘッダは変換できないのでスキップ
33              continue
34
35          temperature = float (daily_data[1]) # 気温データを文字型 -> 数値型に変換
36
37          if max < temperature:               # 最高気温か比較
38              max = temperature               # 最高気温を更新
39              date = daily_data[0]            # 最高気温日を更新
40
41          total = total + temperature         # 気温の合計
42          n = n + 1
43
44  average = total / n                         # 平均気温
45
46  #
47  # 画面表示
48  #
49  print ("日時：", date, ", 最高気温：", max, "℃, 平均気温：", average, "℃\n")
50
51  #
52  # ファイルへの出力
53  #
54  with open ("average.dat", 'w') as w_file:   # ファイルを書き込みモードでオープン
55      #
56      # '+' は文字列の連結, str () は数値型 -> 文字型に変換
57      #
58      w_file.write ("日時：" + date + ", 最高気温：" + str (max) + "␣℃\n")
59      w_file.write ("平均気温：" + str (average) + "␣℃\n")
```

ソースコード 6.10　file_sample.py

ソースコード 6.10 の 27 行目で読み込むファイルを 54 行目で書き込む（新規/上書きモード）ファイルを指定しています.

```
コマンドプロンプト> python file_sample.py
日時： 2015/Aug/28 , 最高気温： 39.968 ℃, 平均気温： 33.88219354838709 ℃

コマンドプロンプト> type average.dat
日時：2015/Aug/28, 最高気温：39.968 ℃
平均気温：33.88219354838709 ℃

コマンドプロンプト>
```

図 6.13　file_sample スクリプトの実行結果

6.7　コマンドライン引数の取り扱い

Python のコマンド実行時に，データやファイルなどの引数与えることができます．これをコマンドライン引数と呼びます．

コマンドライン引数を与えるには，sys モジュールの sys.argv を使用します．

ソースコード 6.11 に，sys.argv によるコマンドライン引数の取得例を示します．

```
1  #!/usr/local/bin/python3
2
3
4  import sys                  # sys モジュールをインポート
5
6  argc = len (sys.argv)       # コマンドラインの引数の数
7  parameters = sys.argv       # コマンドライン引数のリスト
8
9  command = sys.argv[0]       # Python スクリプトファイル名
10
11 if (argc < 2):              # 引数がなかったら Usage を表示して終了
12     print ("Usage:␣", command, "argments␣.....")
13     exit (1)
14
15 else:
16     n = 0
17
18     print ("argc:", argc)
19     print ("parameter:", parameters, "\n")
20
21     for s in parameters:
22         print ("sys.argv[", n, "]:␣", s, sep='')
23         n = n + 1
```

ソースコード 6.11　commandline_sample.py

ソースコード 6.11 のスクリプトの動作確認は図 6.14 に示すように，任意のコマンドライン引数を与えます．

```
コマンドプロンプト> python commandline_sample.py 2015 Aug temperature.dat
argc: 4
parameter: ['commandline_sample.py', '2015', 'Aug', 'temperature.dat']

sys.argv[0]: commandline_sample.py
sys.argv[1]: 2015
sys.argv[2]: Aug
sys.argv[3]: temperature.dat

コマンドプロンプト>
```

図 6.14　commandline_sample スクリプトの実行結果

sys.argv は，図 6.14 に示すようにコマンドライン引数をリストとして取得します．また，コマンドライン引数の数は len 関数で取得することができます．

6.8 Python 組み込み関数，組み込みクラスなど

6.8.1 print 関数

print 関数は，複数の引数 *object, ···* を受け取り，それらを連結して標準出力やファイルに出力します．

書式：print (<*object, ···*><, sep='*s*'><, end='*eoc*'><, file=*file_stream*>)

引数の *object* は数値やリストなど異なった型のオブジェクトを複数記述することができます．

s はセパレータ（デフォルトは，半角スペース）で，オブジェクト *object* を出力する際の区切りに使用されます．また，*eoc* は改行コード（デフォルトは，¥n）で，*object* を出力する際にオブジェクトの末尾に付加されます．

最後に，*file_stream* はファイルストリームで *object* の出力先（デフォルトは，sys.stdout：標準出力）を指定します．

引数はすべて省略可能で，sep='*s*' や end='¥*n*'，file=*file_stream* が省略された場合には，デフォルトの値が使われます．

6.8.2 入力検証用関数

Python の文字列型には，さまざまな入力を検証するためのメソッドがあります．代表的なメソッドを以下に示します．

書式：*object*.isdigit ()

object が数値かどうかを検証します．数値なら「True」を返します．

書式：*object*.isalpha ()

object が英字かどうかを検証します．英字なら「True」を返します．

書式：*object*.isspace ()

object がスペースかどうかを検証します．スペースやタブなら「True」を返します．

なお，これらのメソッドの各文字に対する評価を，表 6.3 に示します．

表 6.3　Python の文字列型の入力検証用関数の判定結果

メソッド名	数字				英字		スペース		タブ
	半角	アラビア	漢字	ローマ	半角	全角	半角	全角	
isdigit	True	True	False	False	False	False	False	False	False
isalpha	False	False	False	True	True	True	False	False	False
isspace	False	False	False	False	False	False	True	True	True

表 6.3 に示すように，各メソッドのみでは,「半角数字のみ」や「半角英字のみ」なのかを判定することはできません．そのため,「半角数字のみ」や「半角英字のみ」なのかを判定する場合には，正規表現を使用するなど工夫する必要があります．

6.8.3　len 関数

len 関数は，オブジェクトの長さ（要素の数）を返します．

書式：len（*object*）

引数の *object* は，シーケンス型オブジェクト（文字列，タプル，またはリスト）かマッピング（辞書）です．

6.8.4　range 関数

range 関数は，指定した条件に従ってリストオブジェクトを生成します．

書式：range（<*start*,> *stop*<, *step*>）

range の引数は全て整数で *start* から *stop* まで *step* ずつ増加するリストオブジェクトを生成します．また，*start* と *step* は省略することができます．*stop* のみを指定した場合には「0 〜 *stop* - *1*」まで 1 ずつ増加するリストオブジェクトを生成します．

6.8.5　書式指定出力 format 関数

format 関数は，文字列の書式を指定して出力することができます.

書式：*target_string*.format (**args*, ***kwargs*)

target_string は書式指定文字列で，中括弧「{ }」に囲まれた置換フィールドを含みます.

引数の **args* や ***kwargs* は，上記の置換フィールドの書式指定文字列に対応するオブジェクトなど，任意の個数の引数を与えることができることを表しています.（ソースコード 6.12，6.14，6.15 参照）

```
1  #!/usr/local/bin/python3
2
3
4  print ("■【新スタイル】format␣関数による文字列フォーマット操作\n")
5
6  #
7  # ◆ index 指定
8  #
9  today = "【NS␣1】今日は␣{0}␣です．"
10 print (today.format ("2015␣年␣8␣月␣7␣日（金）"))
11
12 today_news = "【NS␣1】{0}␣の気温は␣{1}␣℃です．\n"
13 print (today_news.format ("2015␣年␣8␣月␣7␣日（金）", 38.6))
14
15 #
16 # ◆キーワード指定
17 #
18 tomorrow_news = "【NS␣2】{date}␣の気温は␣{temperature}␣℃です．\n"
19 print (tomorrow_news.format (date = "2015␣年␣8␣月␣8␣日（土）", temperature = 44.4))
20
21 #
22 #  ◆辞書型指定
23 #
24 tomorrow_news = "【NS␣3】{key[date]}␣の気温は␣{key[temperature]}␣℃です．\n"
25 dict_data = {"date":"2015␣年␣8␣月␣8␣日（土）", "temperature":44.4}
26 print (tomorrow_news.format (key=dict_data))
```

ソースコード 6.12　format_sample.py

```
コマンドプロンプト> python format_sample.py
■【新スタイル】format 関数による文字列フォーマット操作

【NS 1】今日は 2015 年 8 月 7 日（金） です.
【NS 1】2015 年 8 月 7 日（金） の気温は 38.6 ℃です.

【NS 2】2015 年 8 月 8 日（土） の気温は 44.4 ℃です.

【NS 3】2015 年 8 月 8 日（土） の気温は 44.4 ℃です.

コマンドプロンプト>
```

図 6.15　format_sample スクリプトの実行結果

split 関数

split 関数は，与えられた文字列をセパレータで分割し，分割された単語からなるリストを返します.

書式：*object*.split (<*sep*><, *max_split*>)

sep は単語の区切りとするセパレータを指定します．省略した場合には,「半角スペース」,「タブ」,「改行」がセパレータとして適用されます.

max_split は返される要素数で，省略した場合には文字列全てに対して可能な限りの分割を行います.

int 関数

int 関数は，文字列を数値に変換します.

書式：int (*object*<, *base*>)

int 関数は数値でない *object* だけが与えられると，デフォルトの基数を 10 とした数値に変換します．また，*object* と *base* の両方が与えられると，*object* は基数を *base* とした数値に変換します.

type 関数

type 関数は，引数で指定したオブジェクトの型を返します.

書式：type（*object*）

引数に型を調べたい *object* を記述します.
ソースコード 6.13 にサンプルを示します.

```
1  #!/usr/local/bin/python3
2
3
4  io = 100                                                  # 整数型オブジェクト
5  fo = 3.14159                                              # 浮動小数点型オブジェクト
6  so = "National␣Institute␣of␣Technology."                  # 文字列型オブジェクト
7  lo = ["National", "Institute", "of", "Technology."]       # リストオブジェクト
8  to = ("National", "Institute", "of", "Technology.")       # タプルオブジェクト
9
10 mo = set (['Information', 'Engineering'])                 # set オブジェクト
11 no = frozenset (['Chemical', 'Engineering'])              # frozenset オブジェクト
12 do = {109:"プログラミング␣II", 115:"情報数学␣II"}          # 辞書型オブジェクト
13
14 #
15 # それぞれの「型」を確認する
16 #
17 print ("io␣オブジェクトの型：", type (io))
18 print ("fo␣オブジェクトの型：", type (fo))
19 print ("so␣オブジェクトの型：", type (so))
20 print ("lo␣オブジェクトの型：", type (lo))
21 print ("to␣オブジェクトの型：", type (to))
22 print ("mo␣オブジェクトの型：", type (mo))
23 print ("no␣オブジェクトの型：", type (no))
24 print ("do␣オブジェクトの型：", type (do))
```

ソースコード 6.13　type_sample.py

```
コマンドプロンプト> python type_sample.py
ct1: => <class 'int'>
ct2: => <class 'float'>
ct3: => <class 'str'>
ct4: => <class 'list'>
ct5: => <class 'tuple'>
ct6: => <class 'set'>
ct7: => <class 'frozenset'>
ct8: => <class 'dict'>

コマンドプロンプト>
```

図 6.16　type_sample スクリプトの実行結果

del 文

`del` 文は，引数で指定したオブジェクトやオブジェクトの要素を削除します．

書式：`del` *object*[*index*|*key*]

`del` 文の *object* には削除したいオブジェクトを指定します．*object* がリストオブジェクトの場合，*index* には削除したい要素の「インデックス」や「開始インデックス：終了インデックス」形式で指定します．また，辞書オブジェクトの場合には，削除したい要素の「キー」を指定します．

6.9　Python のモジュール

モジュールとは Python の定義や文が記述されたファイルです．ファイル名はモジュール名を用い，拡張子には「`.py`」とします．
これらのモジュールは `import` 文を使って取り込むことができます．

書式：`import` *module_name*

`import` 文は常にファイルの先頭に記述します．

Python では，表 6.4 に示すようなモジュールが標準で用意されています．(https://docs.python.org/ja/3/py-modindex.html より抜粋)

表 6.4　Python 標準モジュール

モジュール名	モジュールの概要
argparse	コマンドラインオプション，引数，サブコマンドのパーサー
math	数学関数（π などの定数や三角関数など）
os	様々なオペレーティングシステムインタフェース
random	疑似乱数の生成
re	正規表現のパターンマッチング操作を提供
sys	システムパラメータと関数
time	時刻データへのアクセスと変換

6.9.1　擬似乱数の生成

Python では疑似乱数を生成する `random` モジュールが用意されています．`random` モジュールでは，以下に示すようなメソッドを使用することができます．

random.seed メソッド　乱数生成器の初期化をします.

書式：`random.seed (<a=`*n*`>)`

n は疑似乱数を生成する際に使用される数値で，省略された場合や “None” が指定された場合には，現在のシステム時刻（デフォルト）を使用します.

random.randrange メソッド　*start* $\leqq N \leqq$ *stop* までのランダムな整数を返します.

書式：`random.randrange (`*start*`, `*stop*`<, `*step*`>)`

step は増分で省略することができます．*step* を省略した場合，デフォルトの増分は +1 となります.

random.random メソッド　$0.0 \leqq N \leqq 1.0$ までのランダムな浮動小数点を返します.

書式：`random.random ()`

random.uniform メソッド　$a \leqq z$ なら $a \leqq N \leqq z$，$z \leqq a$ なら $z \leqq N \leqq a$ までのランダムな浮動小数点を返します.

書式：`random.uniform (`*a*`, `*z*`)`

random.choice メソッド　引数で指定された空でないシーケンス型オブジェクト *object* の中からランダムに要素を返します.

書式：`random.choice (`*object*`)`

ソースコード 6.14 に疑似乱数の生成を用いたサンプルを示します.

```
1  #!/usr/local/bin/python3
2
3
4  import random                                  # random モジュールの取り込み
5
6  year = "2015"
7  month = "Aug"
8
```

```
 9  #
10  # レコードフォーマットの指定
11  #
12  #   {0} は第 1 番目の仮引数 year
13  #   {1} は第 2 番目の仮引数 month
14  #   {2:02d} は第 3 番目の仮引数 day を '0' 埋めの 2 桁に整形
15  #   {3:2.3f} は第 4 番目の仮引数 temperature を整数部 2 桁, 小数点以下 3 桁に整形
16  #
17  record_style = "{0}/{1}/{2:02d}␣{3:2.3f}"       # {2:02d} と {3:2.3f} の間は ［TAB］
18
19  print ("␣␣年/月/日␣␣␣␣␣␣気温［℃］")            # ヘッダの表示
20  for day in range (1, 32):                       # 1 〜 31 までの繰り返し
21
22      random.seed ()                              # 乱数生成器の初期化
23      temperature = random.uniform (30.0, 40.0)   # 30.0 〜 40.0 の乱数を生成
24
25      print (record_style.format (year, month, day, temperature))
26
27      day = day + 1
```

ソースコード 6.14　random_sample.py

上記ソースコード 6.14 の実行結果を図 6.17 に示します.

```
コマンドプロンプト> python random_sample.py
  年/月/日       気温［℃］
2015/Aug/01     37.814
2015/Aug/02     31.029
2015/Aug/03     37.069
2015/Aug/04     31.190
          :
2015/Aug/28     39.968
2015/Aug/29     31.228
2015/Aug/30     30.830
2015/Aug/31     32.223

コマンドプロンプト>
```

図 6.17　random_sample スクリプトの実行結果

6.10　文法（応用編）

6.10.1　関数 def 文

関数とは，一連の処理をひとまとめにしたもので，関数定義は def 文を使用します.

書式：def *function_name* (<*args, ···*>):
　　statement
　　<return (*return_value* <, *return_value_1*, >···, <*return_value_n*>)>

function_name は定義する関数名で命名規則は変数と同様です.

args は関数の引数（省略可）で，必ず引数を必要とする場合には，単独の変数名 *args* を記述します．省略可能な引数とする場合には，*args* = *default_value* と記述します．さらに定義外の通常の引数をとる場合には，**args* と記述します．定義外の代入型の引数をとる場合には，***args* と記述します．これらの引数の区切りには，変数名をカンマ「,」で区切ります.

関数名の次には，字下げを行い関数で行う処理 *statement* を記述します.

関数での処理結果（値やオブジェクト）を返すには return 文を使用します．return 文に複数の *return_value* を記述するとタプルで返します．また，return 文や *return_value* を省略した場合には，None を指定したことになります.

ソースコード 6.15 に，第 1 引数に数値型，第 2, 3 引数に文字列型の 3 つの引数をとる関数の定義と戻り値，および関数の呼び出し方のサンプルを示します.

```
1  #!/usr/local/bin/python3
2
3
4  #
5  # 変数の設定（初期化）
6  #
7  grade = 3
8  department = "I"
9  subject = "Experiment"
10 title = "Python␣Script"
11 room = "情報工学科␣情報処理実習室"
12 staff = ["Iwata", "K.Yamaguchi", "Okamura", "Nishino"]
13 days = {1:"2016/Jan/15", 2:"2016/Jan/22", 3:"2016/Jan/29", 4:"2016/Feb/05"}
14
15 #
16 # 関数の定義
17 #
18 #   ◆ 引数の型と順番
19 #   i:              数値型 引数（省略不可）
20 #   s1, s2:         文字列型 引数（省略不可）
21 #   chapter = 1:    キーワード型 引数（省略可，デフォルト値：1）
22 #   *args:          上記の引数に該当しない残りの順序引数をタプルで受け取る
23 #   **kwargs:       上記同様 残りのキーワード引数を辞書型で受け取る
24 #
25 #   ◆ 戻り値
26 #   引数で与えられた i を文字列に変換し s1 と結合して，その他の引数も全てタプルで返す.
27 #
28 def function (i, s1, s2, chapter = 1, *args, **kwargs):
29
30     print ("i␣=", i)
```

```
31      print ("s1␣=", s1)
32      print ("s2␣=", s2)
33      print ("chapter␣=", chapter)
34      print ("args␣=", args)
35      print ("kwargs␣=", kwargs, "\n")
36
37      return (str (i) + s1, s2, chapter, args, kwargs)        # 戻り値
38
39  #
40  # 関数の呼び出し
41  #
42  return_value = function (grade, department, subject, 10,
43          title, room, staff, days,
44          cuation="Python␣の仕様は変更されることがあります. ", date="2015/Aug/19")
45
46  print ("return_value␣=", return_value)
```

ソースコード 6.15　def_sample.py

図 6.18 に，ソースコード 6.15 の実行画面を示します.（紙面の都合上，改行しています.）

```
コマンドプロンプト> python def_sample.py
i = 3
s1 = I
s2 = Experiment
chapter = 10
args = ('Python Script', ' 情報工学科 情報処理実習室',
        ['Iwata', 'K.Yamaguchi', 'Okamura', 'Nishino'],
        {1: '2016/Jan/15', 2: '2016/Jan/22',
        3: '2016/Jan/29', 4: '2016/Feb/05'})
kwargs = {'date': '2015/Aug/19',
        'cuation': 'Python の仕様は変更されることがあります. '}

return_value = ('3I', 'Experiment', 10,
        ('Python Script', ' 情報工学科 情報処理実習室',
                ['Iwata', 'K.Yamaguchi', 'Okamura', 'Nishino'],
                {1: '2016/Jan/15', 2: '2016/Jan/22',
                3: '2016/Jan/29', 4: '2016/Feb/05'}),
        {'date': '2015/Aug/19',
                'cuation': 'Python の仕様は変更されることがあります. '})

コマンドプロンプト>
```

図 6.18　def_sample スクリプトの実行結果

6.11　実験

実験に際しては，Tab キーの補完機能や，上キーの入力履歴参照機能をうまく利用して効率的に進めること.

6.11.1　【スクリプトプログラミングの学習】

1. 指導書 44 ページの Python のサンプルスクリプト（ソースコード 6.1）を入力し以下の動作を確認しなさい.

```
Z:\Python> python four_arithmetic.py      # Python スクリプトの実行
10 + 3 = 13
10 - 3 = 7
10 * 3 = 30
10 / 3 = 3.3333333333333335
10 % 3 = 1
10 // 3 = 3

Z:\Python>
```

図 6.19　Python スクリプト（four_arithmetic.py）の実行方法（再）

6.11.2　【インデントと基本構文の習得】

以下に示す条件を満たすプログラムを作成しなさい．指導書 73 ページの `format` を使うとよい.

1. 九九の表を表示（2 重の繰り返し構造を使用すること）
   - 横線 '-' は半角のマイナス（ハイフン）を使うこと.
   - 縦線 '|' は半角のパイプ記号を使うこと.
   - 交差点 '+' は半角のプラスを使うこと.

```
コマンドプロンプト> python k001.py
   |  1  2  3  4  5  6  7  8  9
---+--------------------------
 1 |  1  2  3  4  5  6  7  8  9
 2 |  2  4  6  8 10 12 14 16 18
 3 |  3  6  9 12 15 18 21 24 27
 4 |  4  8 12 16 20 24 28 32 36
 5 |  5 10 15 20 25 30 35 40 45
 6 |  6 12 18 24 30 36 42 48 54
 7 |  7 14 21 28 35 42 49 56 63
 8 |  8 16 24 32 40 48 56 64 72
 9 |  9 18 27 36 45 54 63 72 81
コマンドプロンプト>
```

6.11.3　【基本構文と標準入力の習得】

以下に示す条件を満たすプログラムを作成しなさい．ただし，レポートには課題 2 【日数計算】についてのみアルゴリズムと実験結果を記述しなさい．ほかの課題についてはプログラミングの練習として自主学習に用いなさい.

1. うるう年の判定
   コマンドプロンプト> python k011.py
   何年（YYYY）： 2015
   2015 年は「平年」です.
   コマンドプロンプト>

2. 日数計算
   コマンドプロンプト> python k012.py
   開始日（YYYY/MM/DD）： 2015/08/07
   終了日（YYYY/MM/DD）： 2020/07/24
   日数差： 1813 日
   コマンドプロンプト>

3. 時間計算
   コマンドプロンプト> python k013.py
   開始時間（HH:MM）： 16:15
   終了時間（HH:MM）： 23:30
   経過時間： 7 時間 15 分
   コマンドプロンプト>

6.11.4　【基本構文とファイル操作の習得】

以下に示す条件を満たすプログラムを作成しなさい．ただし，レポートには課題 3 【ソート】についてのみアルゴリズムと実験結果を記述しなさい．ほかの課題についてはプログラミングの練習として自主学習に用いなさい.

1. ファイルの基本操作（1）
   指導書 77 ページのソースコード 6.14 を改良して，結果を temperature.dat ファイルに出力するようにしなさい.
   ファイル出力については，指導書 68 ページの with 文を参考にするとよい．コマンドプロンプトで入力する type 命令は，指定するファイルの中身をコマンドプロンプトで確認するという命令である．もちろんメモ帳などを使って確認してもよい.

```
コマンドプロンプト> python k021.py
コマンドプロンプト> type temperature.dat
2015/Aug/01     36.332
2015/Aug/02     39.767
2015/Aug/03     37.117
          :
2015/Aug/29     30.347
2015/Aug/30     36.842
```

```
2015/Aug/31     34.189
コマンドプロンプト>
```

2. ファイルの基本操作（2）
   指導書 68 ページのソースコード 6.10 を改良して，最低気温とその日時も average.dat ファイルに出力するようにしなさい．
   ただし，出力例のように小数点は 3 桁に固定するよう工夫すること．なお，入力ファイルの temperature.dat については，課題 1 で作成したものを用いること．

```
コマンドプロンプト> python k022.py
コマンドプロンプト> type average.dat
日時： 2015/Aug/29 ，最低気温： 30.347 ℃
日時： 2015/Aug/23 ，最高気温： 39.922 ℃
平均気温： 34.189 ℃
コマンドプロンプト>
```

3. ソート
   temperature.dat ファイルを読み込み，昇順か降順か指定された順にソートするプログラムを作成しなさい．

```
コマンドプロンプト> python k023.py
Data file:  temperature.dat
Rule (昇順:0, 降順:1):  0
 1:  2015/Aug/29     30.347
 2:  2015/Aug/08     30.593
 3:  2015/Aug/11     30.700
             :
29:  2015/Aug/21     39.696
30:  2015/Aug/02     39.767
31:  2015/Aug/23     39.922
コマンドプロンプト>
```

6.11.5　【基本構文とコマンドライン引数の習得】

以下に示す条件を満たすプログラムを作成しなさい．ただし，レポートには課題 2 【ファイルの文字数，単語数，行数のカウント】についてのみアルゴリズムと実験結果を記述しなさい．ほかの課題についてはプログラミングの練習として自主学習に用いなさい．

1. 重複データの削除
   配布する overlap.dat ファイルのうち，同一行があれば削除し，最終的にユニークな行の

みがファイル出力（unique.dat）されるプログラムを作成しなさい．

```
コマンドプロンプト> python k031.py overlap.dat unique.dat
コマンドプロンプト>
```

2. ファイルの文字数，単語数，行数のカウント
   複数のファイルを指定し，それぞれのファイルの文字数，単語数，行数をカウントして，それぞれの結果をファイル出力（プログラム引数の最終指定ファイル名，wc.txt）するプログラムを作成しなさい．ただし，任意の数のファイル数が指定されてもすべて wc.txt に結果が出力できるようにすること．

```
コマンドプロンプト> python k032.py overlap.dat unique.dat ...  wc.txt
コマンドプロンプト> type wc.txt
overlap.dat:  350 文字，164 語，35 行
unique.dat:  108 文字，24 語，8 行
           :
コマンドプロンプト>
```

6.11.6　【総合課題】

以下に示す条件を満たすプログラムを作成しなさい．総合課題については，すべての課題についてアルゴリズム，実験結果をレポートに記述すること．発展課題については，取り組んだ場合に加点する（取り組まなくても減点しない）．

1. アクセスログから特定項目の抽出
   第 5 章の実験（サイバーセキュリティ基礎実験 2）の課題 1 の 5【Web サーバのアクセスログから「アクセス数」「アクセス元の IP アドレス数」を調べなさい】を Python で実現しなさい．また,「SQL コマンドインジェクションを行っていると思われる IP アドレスの一覧」についても出力しなさい．ただし，指導書 76 ページで紹介されている re モジュール（正規表現）を利用すること．

```
コマンドプロンプト> python k042.py access.log
アクセス数：1000
アクセス元の IP アドレス数：10
攻撃元の IP アドレス：(10.0.2.15，10.0.0.1，...)
                  :
コマンドプロンプト>
```

2. Python と Excel の連携
   Web サーバのアクセスログから,「IP アドレスごとのアクセス数」を調べ，第 1 列を IP アドレス，第 2 列をアクセス回数とする CSV 出力しなさい．また，出力した CSV を Excel

で開いて棒グラフ（縦軸アクセス回数，横軸 IP アドレス）を作成（xlsx などで保存しなおす必要があります）し，IP アドレス分布を図示しなさい.

```
コマンドプロンプト> python k043.py access.log ip.csv
コマンドプロンプト> type ip.csv
10.0.2.15, 15
10.0.1.41, 10
         :
コマンドプロンプト>
```

3. 【発展課題】matplotlib を利用したグラフ作成
   棒グラフ（縦軸アクセス回数，横軸 IP アドレス）を Excel を利用せず，`matplotlib` を用いて出力しなさい.

6.12　【参考資料】

6.12.1　Python と Java の違い

Python と Java の違いを表 6.5 〜 表 6.7 に示します.

表 6.5　Python と Java の違い（拡張子，変数など）

		Python	Java
拡張子		`.py`	`.java`
コンパイル		不要	`javac` *java_source_file*`.java`
実行		`python` *script_file*`.py`	`java` *java_class_file*
コメント	行末	`#` *comment string ....*	`//` *comment string ....*

	複数行	`"""`
		comment string ....
		`"""`
			`/*`
			comment string ....
			`*/`

変数	宣言	不要	必須
	使用可能な文字	半角英数字，アンダースコア	
	制約	最初の文字は英字またはアンダースコア，予約語は使用不可	
	大文字・小文字	区別する	
	有効範囲	あり（グローバル，ローカル）	
文	区切り	改行,（セミコロン「;」も可）	セミコロン「;」
	ブロック	インデント（字下げ）	{ 〜 }

try 〜
except 〜
finally 文
		`try:`
		␣␣␣␣*try_statement*
		:
		`except` *Exception* `as e:`
		␣␣␣␣*exception_statement*
		:
		`finally:`
		␣␣␣␣*finally_statement*
		:
			`try {`
			try_statement;
			:
			`} catch (`*Exception* `e) {`
			exception_statement;
			:
			`} finally {`
			finally_statement;
			:
			}

表 6.6　Python と Java の違い（繰り返し，分岐）

for 文	`for` *v_name* `in` *container*:
	␣␣␣␣*statement*
		`for (`*start*; *stop*; *step*`) {`
		　　statement;
		`}`

while 文	*initialization*
	`while` *expression*:
	␣␣␣␣*statement*
	␣␣␣␣*step*
		initialization;
		`while (`*expression*`) {`
		　　statement;
		　　step;
		`}`

do 〜 while 文		*initialization*;
		`do {`
		　　statement;
		　　step;
		`} while (`*expression*`);`

if 文	`if` *expression*:
	␣␣␣␣*statement*
	`elif` *expression_1*:
	␣␣␣␣*statement_1*
	　　:
	`else:`
	␣␣␣␣*else_statement*
		`if (`*expression*`) {`
		　　statement;
		`} else if (`*expression_1*`) {`
		　　statement_1;
		　　:
		`} else {`
		　　else_statement;
		`}`

switch 〜 case 文		`switch (`*expression*`) {`
		　　`case` *const_value_1*:
		　　　　statement_1;
		　　　　break;
		　　　　:
		　　`case` *const_value_n*:
		　　　　statement_n;
		　　　　break;
		　　`default:`
		　　　　default_statement;
		　　　　break;
		`}`

表 6.7　Python と Java の違い（演算子）

		Python	Java
増分演算子			++
減分演算子			--
単項演算子		+ -	+ -
代数演算子	累乗	**	
	乗算	*	*
	除算	/	/
	剰余	%	%
	商の切り捨て	//	
	加算	+	+
	減算	-	-
代入演算子	単純	a = z	a = z
	複合	a **= z a *= z a /= z a %= z a //= z a += z a -= z a <<= z a >>= z a &= z a ^= z a |= z	 a *= z a /= z a %= z a //= z a += z a -= z a <<= z a >>= z a >>>= z a &= z a ^= z a |= z
ビット演算子	反転	~	~
	論理積	&	&
	論理和	|	|
	排他的論理和	^	^
	左シフト	<<	<<
	右シフト	>>	>>
比較演算子		a < z a > z a <= z a >= z a == z, a is z a != z, a <> z, a is not z a in z a not in z	a < z a > z a <= z a >= z a == z a != z
論理演算子	論理積	a and z	a && z
	論理和	a or z	a || z
	論理否定	not a	

6.12.2　Python のエラーメッセージ

表 6.8　Python の主なエラーメッセージ

エラーメッセージ	原因
AttributeError: 'NoneType' object has no attribute 'append'	append メソッドは 'NoneType' オブジェクト（値を返さない）です
AttributeError: '****' object has no attribute '*xxx*'	'****' オブジェクトに *xxx* メソッドや属性はありません
IndentationError: expected an indented	必要なインデントが行なわれていません
IndentationError: expected an indented block	必要なインデントブロックが行なわれていません
IndentationError: unexpected indent	不要なインデントがあります
IndentationError: unindent does not match any outer indentation level	インデントレベルが一致しません（タブとスペースが混在している場合）
IndexError: list index out of range	インデックスがリストの範囲を超えています
IndexError: list assignment index out of range	範囲外のリスト代入インデックスを使っている
NameError: name '*xxx*' is not defined	*xxx* の名前が定義されていません
SyntaxError: EOL while scanning string literal	文字列リテラルを閉じる引用符「"」や「'」がありません
SyntaxError: Non-ASCII character '*¥xc6*' in file *script_file.py* on line *n*, but no encoding declared;	*script_file*.py の *n* 行目に ASCII 以外の文字が使われているのに「エンコード宣言」がありません
SyntaxError: invalid syntax	文法に誤りがあります
SyntaxError: non-keyword arg after keyword arg	キーワード引数のあとにキーワード引数でない引数があります
TabError: inconsistent use of tabs and spaces in indentation	インデントにおいてタブとスペースの一貫性がありません
TypeError: 'int' object is not iterable	int オブジェクトは繰り返し可能なオブジェクト（iterable）ではありません
TypeError: list indices must be integers, not float	リストインデックスは整数でなければなりません
TypeError: 'str' object is not callable	str オブジェクトは関数呼び出しできません
TypeError: 'tuple' object does not support item assignment	タプルオブジェクトの要素の変更はサポートされていません
TypeError: Can't convert 'float' object to str implicitly	浮動小数点型オブジェクトを文字列に変換できません
TypeError: unsupported operand type(s) for `+`: 'int' and 'str' TypeError: unsupported operand type(s) for `/`: 'str' and 'int'	数値型と文字列型の結合はサポートされていません 文字列型と数値型の除算はサポートされていません
ValueError: int () base must be `>=` 2 and `<=` 36	int 関数で指定可能な基数は 2 以上 36 以下でなければなりません

関連図書

[1] Python 3.7.4 ドキュメント, https://docs.python.org/ja/3/, (2019 年 7 月 23 日参照)

[2] Python コードのスタイルガイド https://pep8-ja.readthedocs.io/ja/latest/, (2019 年 7 月 23 日参照)

[3] 有澤 健治 著,「Python によるプログラミング入門」第 6 版, 愛知大学経営学部, http://ar.nyx.link/python/text.pdf, (2019 年 7 月 23 日参照)