Python 実習マニュアル

第零版 revision03

大黒学

Python 実習マニュアル・第零版 revision03
著者——大黒学

2013 年 10 月　8 日（火）　第零版発行
2015 年　1 月 31 日（土）　第零版 revision03 発行

目次

第 1 章　Python の基礎

1.1　プログラム

1.1.1　文書と言語

文字を並べることによって何かを記述したものは、「文書」(document) と呼ばれます。

文書を作るためには、記述したいことを意味として持つように、文字を並べていく必要があります。そして、そのためには、文字をどのように並べればどのような意味になるかということを定めた規則が必要になります。そのような規則は、「言語」(language) と呼ばれます。

人間に読んでもらうことを第一の目的とする文書を書く場合は、日本語や中国語やアラビア語のような、「自然言語」(natural language) と呼ばれる言語が使われます。自然言語というのは、人間の社会の中で自然発生的に形成された言語のことです。

言語には、自然言語のほかに、人間が意図的に設計することによって作られた、「人工言語」(artificial language) と呼ばれるものもあります。人間ではなくてコンピュータに読んでもらうことを第一の目的とする文書を書く場合は、通常、自然言語ではなく人工言語が使われます。

1.1.2　プログラムとプログラミング

コンピュータに何らかの動作を実行させるためには、実行してほしいことがどんな動作なのかということを記述した文書をコンピュータに与える必要があります。そのような文書は、「プログラム」(program) と呼ばれます。

プログラムを作成するためには、プログラムを書くという作業だけではなくて、プログラムの構造を設計したり、プログラムの動作をテストしたり、その不具合を修正したりするというような、さまざまな作業が必要になります。そのような、プログラムを作成するために必要となるさまざまな作業の全体は、「プログラミング」(programming) と呼ばれます。

1.1.3　プログラミング言語

プログラムというのも文書の一種ですから、それを書くためには何らかの言語が必要になります。プログラムを書く場合には、プログラムを書くことだけを目的として作られた人工言語を使うのが普通です。そのような、プログラムを書くための専用の言語は、「プログラミング言語」(programming language) と呼ばれます。

プログラミング言語には、たくさんのものがあります。例を挙げると、Fortran、COBOL、Lisp、Pascal、Basic、C、AWK、Smalltalk、ML、Prolog、Perl、PostScript、Tcl、Java、Ruby、…… というように、枚挙にいとまがないほどです。

1.1.4　スクリプト言語

プログラミング言語を設計するときには、どのようなことを得意とする言語を作るのかという方針を立てる必要があります。複数の方針を立ててプログラミング言語を設計することも可能です。しかし、プログラムがコンピュータによって実行されるときの効率を向上させるという方針と、プログラムが人間にとって書きやすくて読みやすいものになるようにするという方針とは、トレードオフの関係にあります。つまり、それらの二つの方針を両立させることは、とても困難なことなのです。

プログラミング言語の中には、書きやすさや読みやすさよりも実行の効率を優先させて設計されたものもあれば、それとは逆に、効率よりも書きやすさや読みやすさを優先させて設計されたものもあります。後者の方針で設計されたプログラミング言語は、「スクリプト言語」(scripting language) と呼ばれます[1]。スクリプト言語としては、sed、awk、Perl、Tcl、Python、Ruby などがよく使われています。

プログラミング言語で書かれた文書は「プログラム」と呼ばれるわけですが、スクリプト言語で書かれた文書は、「スクリプト」(script) と呼ばれることもあります。

1.1.5　この文章について

この文章（「Python 実習マニュアル」）は、Python というスクリプト言語を使って、プログラムというものの書き方について説明する、ということを目的とするチュートリアルです。

---

[1] スクリプト言語は、「軽量言語」(lightweight language) と呼ばれることもあります。

Python というのは、Guido van Rossum さんという人によって設計されたスクリプト言語です。この言語は、文法が単純ですので、プログラミングの学習にとても適しています。また、「ライブラリ」とよばれる拡張機能が充実していますので、実用性も申し分ありません。

1.2 インタプリタ

1.2.1 言語処理系

コンピュータというものは異質な二つの要素から構成されていて、それぞれの要素は、「ハードウェア」(hardware) と「ソフトウェア」(software) と呼ばれます。ハードウェアというのは物理的な装置のことで、ソフトウェアというのはプログラムなどのデータのことです。

コンピュータは、さまざまなプログラミング言語で書かれたプログラムを理解して実行することができます。しかし、コンピュータのハードウェアが、ソフトウェアの助力を得ないで単独で理解することのできるプログラミング言語は、ハードウェアの種類によって決まっているひとつの言語だけです。

ハードウェアが理解することのできるプログラミング言語は、そのハードウェアの「機械語」(machine language) と呼ばれます。機械語というのは、人間にとっては書くことも読むことも困難な言語ですので、人間が機械語でプログラムを書くということはめったにありません。

人間にとって書いたり読んだりすることが容易なプログラミング言語で書かれたプログラムをコンピュータに理解させるためには、そのためのプログラムが必要になります。そのような、人間が書いたプログラムをコンピュータに理解させるためのプログラムのことを、「言語処理系」(language processor) と呼びます（「言語」を省略して、単に「処理系」と呼ぶこともあります)。

言語処理系には、「コンパイラ」(compiler) と「インタプリタ」(interpreter) と呼ばれる二つの種類があります。コンパイラというのは、人間が書いたプログラムを機械語に翻訳するプログラムのことで、インタプリタというのは、人間が書いたプログラムがあらわしている動作をコンピュータに実行させるプログラムのことです。

1.2.2 Python の処理系

Python の処理系は、Python Software Foundation (PSF) という非営利団体によってオープンソースとして開発されていて、PSF の公式サイトからダウンロードすることができます。

Python Software Foundation の公式サイト　`http://www.python.org/`

ちなみに、PSF が開発している Python の処理系は、コンパイラではなくてインタプリタです。

PSF が開発している Python の処理系は、日本 Python ユーザ会 (Python Japan User's Group, PyJUG) という非営利団体の公式サイトからも、ダウンロードすることができます。

日本 Python ユーザ会の公式サイト　`http://www.python.jp/`

1.2.3 プログラムの入力

プログラムをファイルに保存したり、すでにファイルに保存されているプログラムを修正したりしたいときは、「テキストエディター」(text editor) と呼ばれるソフトを使います（テキストエディターは、単に「エディター」(editor) と呼ばれることもあります)。

それでは、何らかのテキストエディターを使って、次のプログラムを入力して、`hello.py` というファイルに保存してください。

プログラムの例　`hello.py`

```
print('Hello, world!')
```

このプログラムは、実行すると、「Hello, world!」という言葉を出力する、という動作をします。

Python で書かれたプログラムをファイルに保存する場合、ファイル名の拡張子は、`.py` にします。

1.2.4 プログラムの実行

Python で書かれたプログラムは、

`python` パス名

というコマンドによって、Python のインタプリタに実行させることができます。コマンドライン引数として指定するのは、プログラムが保存されているファイルのパス名です。

それでは、先ほど入力したプログラムをインタプリタに実行させてみましょう。コマンドを入力するためのアプリ（Linux や Mac OS ならばターミナル、Windows ならばコマンドプロンプト）を起動して、プログラムのファイルがあるフォルダをカレントフォルダにして、

```
python hello.py
```

というコマンドを入力してみてください。そうすると、インタプリタによってプログラムが実行されて、

```
Hello, world!
```

という言葉が出力されるはずです。

1.2.5 プログラムの文字コード

プログラムというのは、文字でできているデータ、すなわちテキストデータですので、何らかの文字コードを使って書かれることになります。

Python のインタプリタは、デフォルトでは、UTF-8 という文字コードでプログラムが書かれているとみなしてそれを処理します。ですから、日本語の文字を含んでいるプログラムも、それを UTF-8 でファイルに保存すれば、正しく処理されるはずです。

それでは、次のプログラムを UTF-8 でファイルに保存して、Python のインタプリタに実行させてみてください。

プログラムの例 utf8.py

```
print('こんにちは、世界！')
```

1.2.6 エラー

プログラムの中には、何らかの間違いが含まれていることがあります。そのような、プログラムの中に含まれている間違いは、「エラー」(error) と呼ばれます。

言語処理系は、入力されたプログラムがエラーを含んでいる場合、そのエラーについてのメッセージを出力します。そのような、エラーについてのメッセージは、「エラーメッセージ」(error message) と呼ばれます。

それでは、エラーを含んでいる次の Python のプログラムをインタプリタに実行させてみてください。

プログラムの例 error.py

```
brint('Hello, world!')
```

そうすると、Python のインタプリタは、次のようなエラーメッセージを出力します。

```
Traceback (most recent call last):
  File "error.py", line 1, in <module>
    brint('Hello, world!')
NameError: name 'brint' is not defined
```

このエラーメッセージは、brint という名前のものが定義されていないということを教えてくれています。

1.3 インタラクティブシェル

1.3.1 インタラクティブシェルとは何か

前節では、ファイルに保存されているプログラムを Python のインタプリタに実行させる方法について説明したわけですが、この節では、「インタラクティブシェル」(interactive shell) と呼ばれるソフトを使ってプログラムをインタプリタに実行させる方法について説明したいと思います。

インタラクティブシェルというソフトは、

(1) プロンプトを表示する。
(2) プログラムを読み込む。
(3) 読み込んだプログラムをインタプリタに実行させる。

ということを延々と繰り返すように作られています。ですから、インタラクティブシェルを使うことによって、プログラムをキーボードから直接入力して、それをインタプリタに実行させる、ということができます。

1.3.2 インタラクティブシェルの起動

インタラクティブシェルは、コマンドライン引数を何も書かずに、

```
python
```

というコマンドをシェルに入力することによって、起動することができます。

それでは、実際にインタラクティブシェルを起動してみてください。そうすると、

```
>>>
```

というプロンプトが表示されるはずです。

1.3.3 インタラクティブシェルの終了

インタラクティブシェルは、`exit()` と入力するか、または `quit()` と入力することによって終了させることができます。

それでは、実際にインタラクティブシェルを終了させてみてください。インタラクティブシェルが終了すると、Linux や Mac OS の場合はターミナルのプロンプトが、Windows の場合はコマンドプロンプトのプロンプトが表示されます。

1.3.4 式と評価と値

Python のプログラムは、「式」(expression) または「文」(statement) と呼ばれる文字の列から構成されます。式も文も、何らかの動作をあらわしている文字の列です。

式があらわしている動作を実行することを、式を「評価する」(evaluate) と言います。式を評価すると、その結果として 1 個のデータが得られます。式を評価することによって得られたデータは、その式の「値」(value) と呼ばれます。

インタラクティブシェルに対して式を入力すると、インタラクティブシェルはその式を評価して、得られた値を出力します。

1.3.5 リテラル

特定のデータを生成するという動作を記述した式は、「リテラル」(literal) と呼ばれます。リテラルを評価すると、それによって生成されたデータが、そのリテラルの値になります。

リテラルにはさまざまな種類があるのですが、ここでは、整数と文字列のデータを生成するリテラルについて、ごく簡単に説明しておきたいと思います[2]。

0 から 9 までの数字を並べてできる列は、整数のデータを生成するリテラルです。たとえば、2800 というのは、整数のデータを生成するリテラルの一例です。このリテラルを評価すると、2800 という整数のデータが生成されて、そのデータが値として得られます。

それでは、整数のデータを生成するリテラルをインタラクティブシェルに入力してみましょう。たとえば、

```
>>> 2800
```

というように、整数のリテラルを入力して、そののちエンターキーを押してみてください。すると、

```
2800
```

と出力されます。リテラルを評価すると、それによって生成されたデータが値として得られますので、2800 というリテラルを評価した結果が、2800 と出力されたわけです。

文字列のデータを生成するリテラルは、その文字列を引用符 (') で囲んだものです。たとえば、`'namako'` というのは、文字列のデータを生成するリテラルの一例です。このリテラルを評価すると、`namako` という文字列のデータが生成されて、そのデータが値として得られます。

それでは、文字列のデータを生成するリテラルをインタラクティブシェルに入力してみましょう。そうすると、

```
>>> 'Hello, world!'
'Hello, world!'
```

[2] リテラルについては、第 2.1 節で、さらに詳細に説明します。

というように、文字列のリテラルを評価することによって得られた文字列のデータが出力されます。

1.3.6 ファイルに保存されているプログラムの実行

インタラクティブシェルは、キーボードから入力されたプログラムだけではなくて、ファイルに保存されているプログラムをインタプリタに実行させることもできます。

ファイルに保存されているプログラムを、インタラクティブシェルを使ってインタプリタに実行させたいときは、そのファイルのあるフォルダをカレントフォルダにして、

```
from モジュール名 import *
```

という文をインタラクティブシェルに入力します。この中の「モジュール名」のところには、プログラムが保存されているファイルの名前から拡張子を取り除いたものを書きます。

それでは、前節で hello.py というファイルに保存したプログラムを、インタラクティブシェルを使ってインタプリタに実行させてみましょう。

hello.py のあるフォルダをカレントフォルダにして、インタラクティブシェルを起動して、

```
from hello import *
```

という文をインタラクティブシェルに入力してみてください。そうすると、

```
>>> from hello import *
Hello, world!
>>>
```

というように、プログラムが実行されるはずです。

1.4 オブジェクト

1.4.1 オブジェクトとは何か

第 1.3.4 節で、「式を評価すると、その結果として 1 個のデータが得られます」と書きましたが、厳密に言うと、これは正しくありません。実は、Python では、式を評価することによって得られるものは、単なるデータではなくて、「オブジェクト」(object) と呼ばれるものなのです。

オブジェクトというのは、箱のようなものだと考えることができます。

オブジェクトという箱の中には、2 種類のものが詰め込まれています。ひとつの種類はデータで、もうひとつの種類は、「メソッド」(method) と呼ばれるものです。

1.4.2 メソッド

メソッドは、何らかの仕事をすることができます。メソッドの仕事というのは、基本的には、自分が所属しているオブジェクトの中にあるデータの操作です。ひとつのオブジェクトはいくつかのメソッドを持っていて、それぞれのメソッドは、自分に固有の仕事を実行します。ですから、オブジェクトというのは、自分の中にあるデータを操作するためのさまざまな機能を持っている箱のことだと考えることができます。

オブジェクトにはさまざまな種類があります。たとえば文字列のオブジェクトや整数のオブジェクトなどです。オブジェクトがどんなメソッドを持っているかというのは、そのオブジェクトの種類ごとに決まっています。たとえば、文字列のオブジェクトは、小文字を大文字に変換するメソッドや、文字列の検索をするメソッドや、指定された区切り文字列で文字列を区切るメソッドなどを持っています。

それぞれのメソッドは、自分を識別するための名前を持っています。たとえば、文字列のオブジェクトが持っている、小文字を大文字に変換するメソッドは、upper という名前を持っています。

1.4.3 メソッドを呼び出す式

メソッドに仕事をさせることを、メソッドを「呼び出す」(call) と言います。

メソッドを呼び出すためには、そのための式を評価する必要があります。メソッドを呼び出す式は、「メソッド呼び出し」(method call) と呼ばれます。

文字列のオブジェクトが持っている upper というメソッドは、

```
exp.upper()
```

というメソッド呼び出しを書くことによって呼び出すことができます。この式の中の *exp* という部分には、文字列のオブジェクトが値として得られる式を書きます。たとえば、

```
'namako'.upper()
```

というメソッド呼び出しを書くことによって、namako という文字列を大文字に変換した結果を求めることができます。

それでは、インタラクティブシェルを使って、文字列のオブジェクトが持っている upper というメソッドを呼び出してみてください。

実行例

```
>>> 'namako'.upper()
'NAMAKO'
```

1.5　空白と改行と注釈

1.5.1　空白

空白という文字（スペースキーを押したときに入力される文字）は、Python のプログラムの意味に影響を与えません（ただし、行の先頭の空白には意味があります）。たとえば、

```
print('Hello, world!')
```

というプログラムは、

```
print ( 'Hello, world!'  )
```

と書いたとしても同じ意味になりますし、

```
print   (   'Hello, world!'   )
```

と書いたとしても同じ意味になります。

ただし、文字列のリテラルの中に空白を挿入した場合は、その空白を含んだ文字列のオブジェクトが生成されます。たとえば、

```
print('H e l l o ,  w o r l d !')
```

というプログラムは、

```
H e l l o ,  w o r l d !
```

という文字列を出力します。

名前の途中には、空白を挿入することができません。ですから、print という名前を、

```
p r i n t
```

と書くことはできません。

1.5.2　改行

Python では、原則としては式の途中に改行（エンターキーを押したときに入力される文字）を挿入することはできません。ただし、丸括弧などの括弧類で囲まれた部分は、その途中に改行を挿入してもかまいません。その場合、空白の場合と同じように、改行を挿入したとしてもプログラムの意味には影響を与えません。たとえば、

```
print('Hello, world!')
```

というプログラムは、全体が 1 個の式になっていますが、一部分が丸括弧で囲まれていますので、その部分に改行を挿入して、

```
print(
'Hello, world!'
)
```

と書いたとしても、同じ意味になります。

インタラクティブシェルには、改行を含む式や文を入力することも可能です。式または文の途中で改行が入力された場合、インタラクティブシェルは、式または文の入力がまだ終わっていないということを示すために、

```
...
```

というプロンプトを表示します。たとえば、上のプログラムをインタラクティブシェルに入力したとすると、

```
>>> print(
... 'Hello, world!'
... )
Hello, world!
>>>
```

というようにプロンプトが変化します。
改行も、名前の途中に挿入することができないという点は、空白と同じです。

1.5.3 注釈

プログラムを書いているとき、それを読む人間（プログラムを書いた人自身もその中に含まれます）に伝えたいことを、そのプログラムの一部分として書いておきたい、ということがしばしばあります。プログラムの中に書かれたそのような文字列は、「注釈」(comment) と呼ばれます。
注釈は、プログラムを処理するプログラムが、「ここからここまでは注釈だ」ということを認識することができるように、注釈を書くための文法にしたがって書く必要があります。
Python では、「この部分は注釈です」ということを言語処理系に知らせるために、井桁 (`#`) という文字を使います。プログラムの中に井桁を書くと、Python の言語処理系は、その直後から改行までの部分を注釈とみなして無視します。
それでは、インタラクティブシェルを使って、井桁から改行までの部分が本当に無視されるかどうかを確かめてみましょう。

```
>>> print('Hello, world!') # I am a comment.
Hello, world!
```

このように、入力したプログラムの中にある「I am a comment.」という部分は、井桁と改行のあいだに書かれていますので、Python のインタプリタはその部分を注釈とみなして無視します。
逆に、井桁を書かなかった場合は、どうなるでしょうか。

```
>>> print('Hello, world!') I am a comment.
  File "<stdin>", line 1
    print('Hello, world!') I am a comment.
                           ^
SyntaxError: invalid syntax
```

このように、井桁を書かなかった場合、インタプリタは、書かれたものをすべて解釈しようとしますので、エラーメッセージが表示されることになります。

1.5.4 コメントアウトとアンコメント

プログラムを作成したり修正したりしているとき、その一部分を一時的に無効にしたい、ということがしばしばあります。そのような場合、無効にしたい部分を削除してしまうと、それを復活させるのに手間がかかりますので、削除するのではなくて、注釈にすることによって無効にするという手段が、しばしば使われます。記述の一部分を注釈にすることによって、それを無効にすることを、その部分を「コメントアウトする」(comment out) と言います。逆に、コメントアウトされている部分を復活させることを、その部分を「アンコメントする」(uncomment) と言います。

第 2 章　式

2.1　リテラル

2.1.1 リテラルの基礎

「リテラル」というものについては、すでに第 1.3.5 項で簡単に説明しましたが、そこでの説明は概略にすぎないものでしたので、この節では、リテラルについて、もう少し詳細に説明したいと思います。
特定のオブジェクトを生成するという動作を記述した式は、「リテラル」(literal) と呼ばれます。たとえば、`'suzume'` のような、文字列を引用符で囲んだものは、リテラルの一種です。

リテラルを評価すると、それによって生成されたオブジェクトが、その値として得られます。たとえば、'suzume'というリテラルを評価すると、それによって生成されたsuzumeという文字列のオブジェクトが、その値として得られます。

なお、この文章のこれから先の部分では、誤解のおそれがない場合、「○○のオブジェクト」のことを単に「○○」と呼ぶことがあります。たとえば、文字列そのものではなくて文字列のオブジェクトのことを指しているということが文脈から明らかに分かる場合には、文字列のオブジェクトのことを単に「文字列」と呼ぶことがあります。

2.1.2 整数リテラル

整数のオブジェクトを生成するリテラルは、「整数リテラル」(integer literal) と呼ばれます。

整数リテラルは、次の 4 種類に分類することができます。

- 10 進数リテラル (decimal integer literal)
- 16 進数リテラル (hexadecimal integer literal)
- 8 進数リテラル (octal integer literal)
- 2 進数リテラル (binary integer literal)

これらの整数リテラルの相違点は、その名前が示しているとおり、整数を表現するための基数です。つまり、それぞれの整数リテラルは、10、16、8、2 のそれぞれを基数として整数を表現します。

10 進数リテラルは、481 とか 3007 というような、数字だけから構成される列（ただし、先頭の数字は 0 以外でないといけません）です。たとえば、481 という 10 進数リテラルを評価すると、481 というプラスの整数が値として得られます。

8 進数リテラル、16 進数リテラル、2 進数リテラルは、基数を示す接頭辞を先頭に書くことによって作られます。基数を示す接頭辞は、8 進数は0o、16 進数は0x、2 進数は0bです（o、x、bは大文字でもかまいません)。たとえば、0o377、0xff、0b11111111は、いずれも、255 という整数を生成します。

2.1.3 浮動小数点数リテラル

ひとつの数値を、数字の列と小数点の位置という二つの要素で表現しているデータは、「浮動小数点数」(floating point number) と呼ばれます。

浮動小数点数のオブジェクトを生成するリテラルは、「浮動小数点数リテラル」(floating point literal) と呼ばれます。

.003、41.56、723.というような、ドット (.) という文字を 1 個だけ含んでいる数字の列は、浮動小数点数のオブジェクトを生成するリテラルになります。この場合、ドットは小数点の位置を示します。

浮動小数点数を生成するリテラルとしては、

a e b

という形のものを書くことも可能です（e は大文字でもかまいません)。a のところには、ドットを 1 個だけ含むかまたは含まない数字の列を書くことができて、b のところには、数字の列または左側にマイナスのある数字の列を書くことができます。この形のリテラルを評価すると、

$$a \times 10^b$$

という浮動小数点数が生成されます。

リテラル	意味
3e8	3×10^8
6.022e23	6.022×10^{23}
6.626e-34	6.626×10^{-34}

2.1.4 マイナスの数値を生成する式

マイナス (-) という文字を書いて、その右側に整数または浮動小数点数のリテラルを書くと、その全体は、マイナスの数値を生成する式になります。たとえば、-56 という式はマイナスの 56 という整数を生成して、-0xff はマイナスの 255 という整数を生成して、-8.317 はマイナスの 8.317 という浮動小数点数を生成します。

マイナスの数値を生成するこのような式の先頭に書かれるマイナスという文字は、リテラルの一部分ではなくて、第 2.2 節で説明することになる「演算子」と呼ばれるものです。

2.1.5 文字列リテラル

文字列のオブジェクトを生成するリテラルは、「文字列リテラル」(string literal) と呼ばれます。
文字列リテラルは、次の 3 種類に分類することができます。

- 短文字列リテラル (short string literal)
- 長文字列リテラル (long string literal)
- 未加工文字列リテラル (raw string literal)

2.1.6 短文字列リテラル

短文字列リテラル (short string literal) は、引用符 (') または二重引用符 (") で文字列を囲むことによって作られます。たとえば、'namako' や "namako" は、短文字列リテラルです。
短文字列リテラルは、引用符または二重引用符で囲まれた中にある文字列を生成します。たとえば、'namako' という短文字列リテラルは、namako という文字列を生成します。

2.1.7 エスケープシーケンス

文字の中には、たとえばビープ音や改ページのように、そのままでは文字列リテラルの中に書くことができない特殊なものがあります。
そのような特殊な文字を文字列リテラルの中に書きたいときに使われるのが、「エスケープシーケンス」(escape sequence) と呼ばれる文字列です。エスケープシーケンスは、必ず、バックスラッシュ(\) という文字で始まります[1]。
エスケープシーケンスには、次のようなものがあります。

\a	ビープ音	\f	改ページ
\t	水平タブ	\v	垂直タブ
\'	引用符	\"	二重引用符
\n	改行	\r	キャリッジリターン
\b	バックスペース	\\	バックスラッシュ
ooo	8 進数 *ooo* に対応する文字	\x*hh*	16 進数 *hh* に対応する文字

短文字列リテラルまたは長文字列リテラルの中にエスケープシーケンスが含まれていた場合、そのエスケープシーケンスは、それが意味している文字に置き換わります。

```
>>> print('namako\numiushi\nkurage')
namako
umiushi
kurage
```

2.1.8 長文字列リテラル

長文字列リテラル (long string literal) は、引用符または二重引用符を 3 個連続して並べたもの（''' または """）で文字列を囲むことによって作られます。たとえば、'''namako''' や """namako""" は、長文字列リテラルです。
長文字列リテラルは、3 連続の引用符または 3 連続の二重引用符で囲まれた中にある文字列を生成します。たとえば、'''namako''' という長文字列リテラルは、namako という文字列を生成します。
短文字列リテラルと長文字列リテラルとの相違点は、改行を含む文字列を生成するための方法です。短文字列リテラルの場合、改行を含む文字列を生成するためには、\n というエスケープシーケンスを書く必要がありますが、長文字列リテラルを使えば、エスケープシーケンスを書かなくても、改行を挿入したいところに改行を書くだけで、改行を含む文字列を生成することができます。ですから、長文字列リテラルは、改行を何個も含む文字列を生成したいときに便利です。

```
>>> print('''namako
... umiushi
... kurage''')
```

[1] バックスラッシュは、日本語の環境では円マーク (¥) で表示されることがあります。

```
namako
umiushi
kurage
```

2.1.9 未加工文字列リテラル

未加工文字列リテラル (raw string literal) は、短文字列リテラルまたは長文字列リテラルの先頭に r または R を書くことによって作られます。たとえば、r'namako' や r"""namako""" は、未加工文字列リテラルです。

未加工文字列リテラルも、短文字列リテラルや長文字列リテラルと同様に、引用符または二重引用符で囲まれた中にある文字列を生成します。たとえば、r'namako' という未加工文字列リテラルは、namako という文字列を生成します。

短文字列リテラルや長文字列リテラルと、未加工文字列リテラルとの相違点は、バックスラッシュが特別扱いされるかどうかです。短文字列リテラルや長文字列リテラルの中に含まれているバックスラッシュは、エスケープシーケンスの 1 文字目として解釈されます。ただし、バックスラッシュから始まる文字列をエスケープシーケンスとして解釈することができなかった場合、バックスラッシュは、それ自身として解釈されます。それに対して、未加工文字列リテラルの場合、その中に含まれているバックスラッシュは特別扱いされることなく、無条件にそれ自身として解釈されます。ですから、未加工文字列リテラルは、Windows のパス名のような、バックスラッシュを何個も含む文字列を生成したいときに便利です。

```
>>> print(r'c:\prog\tex\bin')
c:\prog\tex\bin
```

2.2 演算子

2.2.1 演算子の基礎

Python では、頻繁に必要となる単純な動作は、「演算子」(operator) と呼ばれるものを書くことによって記述することができます。

演算子を使いたいときは、演算子と式とを組み合わせた式を書きます。大多数の演算子は、それと式とを組み合わせた式の構造によって、次の 2 種類に分類することができます。

- 二項演算子 (binary operator)
- 単項演算子 (unary operator)

2.2.2 二項演算子

「二項演算子」(binary operator) と呼ばれるグループに所属している演算子は、

式 二項演算子 式

という構造の式を作ります。

二項演算子を含む式を評価すると、原則的には、まず演算子の左右の式が評価されて、それらの式の値に対して、二項演算子があらわしている動作が実行されて、その結果が式全体の値になります。

2.2.3 算術演算子

数値に対する計算をあらわしている演算子は、「算術演算子」(arithmetic operator) と呼ばれます。

二項演算子でかつ算術演算子であるような演算子としては、次のようなものがあります。

a + b　　a と b とを足し算（加算）する。
a - b　　a から b を引き算（減算）する。
a * b　　a と b とを掛け算（乗算）する。
a / b　　a を b で割り算（除算）する。整数の範囲で割り切れない場合は小数点以下も求める。
a // b　　a を b で割り算（除算）する。小数点以下は切り捨て。
a % b　　a を b で除算したあまりを求める。
a ** b　　a の b 乗（べき乗）を求める。

```
>>> 30+7
37
>>> 30-7
23
>>> 30*7
210
>>> 30/7
4.285714285714286
>>> 30//7
4
>>> 30%7
2
>>> 3**4
81
```

2.2.4 文字列の連結

+という演算子は、数値の加算だけではなくて、文字列の連結という意味も持っています。

+の左右に書かれた式の値の両方が文字列だった場合、+は、数値の加算という動作ではなくて、文字列の連結という動作をします。

```
>>> 'kitsune' + 'udon'
'kitsuneudon'
```

ちなみに、+の左右に書かれた式の値の一方が文字列で、他方が数値だった場合は、エラーになります。

*という演算子は、数値の乗算だけではなくて、指定された回数だけ同じ文字列を連結するという意味も持っています。

*の左右に書かれた式の一方の値が文字列で、他方が整数だった場合、*は、数値の乗算という動作ではなくて、整数で指定された回数だけ文字列を連結するという動作をします。

```
>>> 'hoge' * 6
'hogehogehogehogehogehoge'
```

2.2.5 優先順位

ひとつの式の中に2個以上の演算子が含まれている場合、その式はどのように解釈されるのでしょうか。たとえば、

```
2+3*4
```

という式は、

```
[2+3]*4
```

という構造なのでしょうか。それとも、

```
2+[3*4]
```

という構造なのでしょうか。

この問題は、個々の演算子が持っている「優先順位」(precedence) と呼ばれるものによって解決されます。

優先順位というのは、演算子が左右の式と結合する強さのことだと考えることができます。優先順位が高い演算子は、それが低い演算子よりも、より強く左右の式と結合します。

*と/と//と%は、+と-よりも高い優先順位を持っています。ですから、

```
2+3*4
```

という式は、

```
2+[3*4]
```

という構造だと解釈されます。

```
>>> 2+3*4
14
```

さらに、**は、*と/と//と%よりも高い優先順位を持っています。ですから、

```
2*3**4
```

という式は、

```
2*[3**4]
```

という構造だと解釈されます。

```
>>> 2*3**4
162
```

2.2.6　結合規則

ひとつの式の中に同じ優先順位を持っている 2 個以上の演算子が含まれている場合、その式はどのように解釈されるのでしょうか。たとえば、

```
10-5+2
```

という式は、

```
[10-5]+2
```

という構造なのでしょうか。それとも、

```
10-[5+2]
```

という構造なのでしょうか。

この問題は、同一の優先順位を持つ演算子が共有している「結合規則」(associativity) と呼ばれる性質によって解決されます。

結合規則には、「左結合」(left-associativity) と「右結合」(right-associativity) という二つのものがあります。左結合というのは、左右の式と結合する強さが左にあるものほど強くなるという性質で、右結合というのは、それが右にあるものほど強くなるという性質です。

`+`、`-`、`*`、`/`、`//`、`%` の結合規則は、左結合です。したがって、

```
10-5+2
```

という式は、

```
[10-5]+2
```

という構造だと解釈されます。

```
>>> 10-5+2
7
```

`**` の結合規則は、右結合です。したがって、

```
2**3**4
```

という式は、

```
2**[3**4]
```

という構造だと解釈されます。

```
>>> 2**3**4
2417851639229258349412352
```

2.2.7　丸括弧

ところで、2 と 3 とを足し算して、その結果と 4 とを掛け算したい、というときは、どのような式を書けばいいのでしょうか。先ほど説明したように、`+` と `*` とでは、`*` のほうが優先順位が高くなっていますので、

```
2+3*4
```

と書いたのでは、期待した結果は得られません。

演算子の優先順位や結合規則に縛られずに、自分が望んだとおりに式を解釈してほしい場合は、ひとまとまりの式だと解釈してほしい部分を、丸括弧 (`()`) で囲みます。そうすると、丸括弧で囲まれている部分は、演算子の優先順位や結合規則とは無関係に、ひとまとまりの式だと解釈されます。

```
>>> (2+3)*4
20
>>> (2*3)**4
1296
>>> 10-(5+2)
3
>>> (2**3)**4
4096
```

第 1.5.2 項で説明したように、Python では、原則としては式の途中に改行を挿入することができません。しかし、式の途中にどうしても改行を挿入したいということも、しばしばあります。

途中に改行が挿入された式は、丸括弧を使うことによって書くことができます。なぜなら、Python では、丸括弧などの括弧類で囲まれた部分は、その途中に改行が挿入されていてもかまわないからです。

```
>>> (700+
...   60+
...    5)
765
```

2.2.8 単項演算子

「単項演算子」(unary operator) と呼ばれるグループに所属している演算子は、

単項演算子 式

という構造の式を作ります。

ほとんどの二項演算子は、単項演算子よりも低い優先順位を持っていますが、べき乗を求める二項演算子は、単項演算子よりも高い優先順位を持っています。

2.2.9 符号の反転

- という単項演算子は、数値の符号（プラスかマイナスか）を反転させる算術演算子です。

```
>>> -(3+5)
-8
>>> -(3-5)
2
>>> -3**2
-9
>>> (-3)**2
9
```

2.3 呼び出し

2.3.1 呼び出しとは何か

Python では、動作させることのできるオブジェクトのことを「呼び出し可能オブジェクト」(callable object) と呼びます。

呼び出し可能オブジェクトを動作させることを、それを「呼び出す」(call) と言います。そして、呼び出し可能オブジェクトを呼び出すという動作をあらわす式は、「呼び出し」(call) と呼ばれます。

2.3.2 呼び出し可能オブジェクトの分類

呼び出し可能オブジェクトは、次の 3 種類に分類することができます。

- 関数 (function)
- メソッド (method)
- クラス (class)

関数というのは、もっとも普通の呼び出し可能オブジェクトです。

メソッドというのは、第 1.4.2 項で説明したように、オブジェクトの中にあって、何らかの仕事をするもののことです。

クラスというのは、オブジェクトを生成するという動作をする呼び出し可能オブジェクトのことです。

2.3.3 引数と戻り値

呼び出し可能オブジェクトは、自分が動作を開始する前に、自分を呼び出した者からオブジェクトを受け取ることができます。呼び出し可能オブジェクトが受け取るオブジェクトは、「引数」(argument) と呼ばれます（「引数」は「ひきすう」と読みます）。呼び出し可能オブジェクトは、引数を何個でも受け取ることができます。

呼び出し可能オブジェクトは、自分の動作が終了したのちに、自分を呼び出した者にオブジェクトを返すことができます。呼び出し可能オブジェクトが返すオブジェクトは、「戻り値」(return value) と呼ばれます。呼び出し可能オブジェクトは、引数は何個でも受け取ることができるのに対して、返すことができる戻り値は 1 個だけです。

Python の処理系には、`len` という関数が組み込まれています。この関数は、1 個のオブジェクトを引数として受け取って、その長さ（文字列ならば文字の個数）を求めるという動作をして、その結果を戻り値として返します。たとえば、`umiushi` という文字列を引数として `len` に渡したとすると、`len` は、7 という整数を戻り値として返します。

2.3.4 呼び出しの書き方

呼び出しは、

式 (式, ···)

と書きます。先頭の式は、呼び出し可能オブジェクトが値として得られるものでないといけません。

呼び出しを評価すると、その先頭に書かれた式を評価することによって得られた呼び出し可能オブジェクトが呼び出されて、丸括弧の中に書かれた式の値が、その呼び出し可能オブジェクトに引数として渡されます。そして、その呼び出し可能オブジェクトが返した戻り値が、呼び出しの値になります。

`len` という関数を呼び出して、`umiushi` という文字列を引数として渡す呼び出しは、

```
len('umiushi')
```

と書くことができます。この式を評価すると、その値として 7 という整数が得られます。

```
>>> len('umiushi')
7
```

この例のように、関数に名前が与えられている場合、その名前を評価すると、その関数が値として得られます。

2.3.5 メソッドを求める式

メソッドを求めたいときは、

式 . メソッド名

という形の式を書きます。この形の式を評価すると、ドット (.) の左側の式を評価することによって得られたオブジェクトが持っている、メソッド名で指定されたメソッドが、値として得られます。たとえば、

```
'isoginchaku'.find
```

という式を書くことによって、`isoginchaku` という文字列が持っている `find` というメソッドを求めることができます。

文字列が持っている `find` というメソッドは、自身（メソッドを持っている文字列自身）の一部分として引数が含まれているならば、その位置をあらわす整数を戻り値として返します。自身の一部分として引数が含まれていない場合は、−1 を返します。位置をあらわす整数というのは、先頭の文字を 0 番目と数えて何番目にあるかということです。

```
>>> 'isoginchaku'.find('cha')
6
>>> 'isoginchaku'.find('myu')
-1
```

メソッドの動作を説明するときは、しばしば、そのメソッドを持っているオブジェクトに言及する必要が生じます。このチュートリアルでは、メソッドを持っているオブジェクトに言及したいときは、先ほどのように、「自身」という言葉を使うことにしたいと思います。

2.3.6 ユーザー定義関数と組み込み関数

Python では、「関数定義」(function definition) と呼ばれる文をプログラムの中に書くことによって、関数を自由に定義することができます（関数定義の書き方については、第 3.3 節で説明することにしたいと思います)。プログラムの中に関数定義を書くことによって定義された関数は、「ユーザー定義関数」(user-defined function) と呼ばれます。

「ユーザー定義関数」の対義語は、「組み込み関数」です。「組み込み関数」(built-in function) というのは、Python の処理系に組み込まれている関数のことです。組み込み関数は、関数定義を書かなくても、呼び出しを書くだけで呼び出すことができます。先ほど紹介した `len` も、組み込み関数のひとつです。

2.3.7 読み込み

`input` という組み込み関数は、キーボードから 1 行の文字列を読み込んで、読み込んだ文字列を戻り値として返します（行の末尾の改行は、戻り値には含まれません)。

```
>>> input()
katatsumuri
'katatsumuri'
```

引数として 1 個の文字列を `input` に渡すと、 `input` は、その引数をプロンプトとして出力します。

```
>>> input('What is your name? ')
What is your name? Togawa Arika
'Togawa Arika'
```

2.3.8 出力

`print` という組み込み関数は、何個かのオブジェクトを引数として受け取って、それらのオブジェクトを出力します（文字列以外のオブジェクトは、文字列に変換されて出力されます)。

```
>>> print('Matsubue Takaomi')
Matsubue Takaomi
```

`print` は、2 個以上の引数を受け取ることもできます。2 個以上の引数を受け取った場合、`print` は、それらの引数を空白で区切って出力します。

```
>>> print(38, 27, 64, 58, 80)
38 27 64 58 80
```

`print` は、 `None` というオブジェクトを戻り値として返します。

```
>>> print(print('namako'))
namako
None
```

`None` は、「何も存在しない」という意味で使われるオブジェクトです。インタラクティブシェルは、入力された式の値が `None` だった場合、その式の値を出力しません。

2.3.9 キーワード引数

2 個以上の引数を呼び出し可能オブジェクトに渡すときには、呼び出しの丸括弧の中に、正しい順序で式を書く必要があります。たとえば、`print` を使って 2 個以上のオブジェクトを出力する場合、それらのオブジェクトを求める式は、それらを出力する順序のとおりに呼び出しの中に書かないといけません。

しかし、呼び出し可能オブジェクトに引数を渡す方法としては、順番が意味を持つ通常の方法だけではなくて、それとは異なる別の方法もあります。それは、「キーワード引数」(keyword argument) と呼ばれるものを渡すという方法です。

キーワード引数というのは、オブジェクトとともに、そのオブジェクトが何を意味しているのかということを示すキーワードを呼び出し可能オブジェクトに渡す、というタイプの引数のことです。呼び出し可能オブジェクトにキーワード引数を渡したいときは、呼び出しの中に、

```
キーワード = 式
```

という形のものを書きます。

`print` という関数は、次のようなキーワード引数を受け取ることができます。

sep 複数のオブジェクトを出力する場合に、それらを区切る文字列。デフォルト値は 1 個の空白。

end 最後に出力する文字列。デフォルト値は 1 個の改行。

「デフォルト値」(default value) というのは、引数を受け取らなかった場合に引数の代わりに使われるオブジェクトのことです。

キーワード引数は、普通の引数とは違って、書く順番は自由です。それらがどんな順番で書かれていたとしても、呼び出し可能オブジェクトはそれらを正しく受け取ることができます。ただし、普通の引数とキーワード引数とを 1 個の呼び出しの中に混在させる場合は、まず最初に普通の引数を書いて、そののちにキーワード引数を書く、という順番を守る必要があります。

```
>>> print(2, 3, 5, 7, 11, 13, sep=', ', end=', etc.\n')
2, 3, 5, 7, 11, 13, etc.
>>> print(2, 3, 5, 7, 11, 13, end=', etc.\n', sep=', ')
2, 3, 5, 7, 11, 13, etc.
```

2.3.10 ユーザー定義クラスと組み込み型

Python では、「クラス定義」(class definition) と呼ばれる文をプログラムの中に書くことによって、クラスを自由に定義することができます（クラス定義の書き方については、第 8.2 節で説明することにしたいと思います)。プログラムの中にクラス定義を書くことによって定義されたクラスは、「ユーザー定義クラス」(user-defined class) と呼ばれます。

「ユーザー定義クラス」の対義語は、「組み込み型」です。「組み込み型」(built-in type) というのは、Python の処理系に組み込まれているクラスのことです。組み込み型は、クラス定義を書かなくても、呼び出しを書くだけで呼び出すことができます。

2.3.11 整数を生成する組み込み型

整数は、int という組み込み型から生成されるオブジェクトです。int を明示的に呼び出して、整数をあらわしている文字列を引数として渡すと、int は、その文字列によってあらわされている整数を生成して、その整数を戻り値として返します。

```
>>> int('801')
801
```

int は、引数として浮動小数点数を受け取ることもできます。その場合、int は、引数の小数点以下を切り捨てた整数を生成します。

```
>>> int(6.82)
6
```

2.3.12 浮動小数点数を生成する組み込み型

浮動小数点数は、float という組み込み型から生成されるオブジェクトです。float を明示的に呼び出して、浮動小数点数をあらわしている文字列を引数として渡すと、float は、その文字列によってあらわされている浮動小数点数を生成して、その浮動小数点数を戻り値として返します。

```
>>> float('3.14')
3.14
```

float は、引数として整数を受け取ることもできます。その場合、float は、引数と等しい浮動小数点数を生成します。

```
>>> float(3)
3.0
```

2.3.13 文字列を生成する組み込み型

文字列は、str という組み込み型から生成されるオブジェクトです。str を明示的に呼び出して、任意のオブジェクトを引数として渡すと、str は、そのオブジェクトをあらわす文字列を生成して、その文字列を戻り値として返します。

```
>>> str(801)
'801'
>>> str(3.14)
```

```
'3.14'
```

2.3.14 指定された基数による数値から文字列への変換

引数として数値を渡して str を呼び出すと、 str は、それをあらわす 10 進数を生成して、それを戻り値として返します。 str では、10 以外の基数を使って数値を文字列に変換するということはできません。10 以外の基数を使いたい場合は、 format という組み込み関数を使う必要があります。

format は、1 個目の引数として数値、2 個目の引数として「書式指定」(format specification) と呼ばれる文字列を受け取って、書式指定によって指定された基数を使って 1 個目の引数を文字列に変換して、その結果を戻り値として返します。

書式指定は、16 進数に変換したい場合は x または X、8 進数に変換したい場合は o、2 進数に変換したい場合は b と書きます。

```
>>> format(255, 'x')
'ff'
>>> format(255, 'o')
'377'
>>> format(255, 'b')
'11111111'
```

2.4 式文

2.4.1 式文の基礎

Python では、ひとつの式は、それ自体でひとつの文になることができます。

文として書かれた式は、「式文」(expression statement) と呼ばれます。

式文が意味している動作は、それの式としての動作と同じです。たとえば、

```
print('Masuda Hiromi')
```

という文を実行すると、 Masuda Hiromi という文字列が出力されます。

2.4.2 プログラムの実行

Python のプログラムは、いくつかの文を改行で区切って並べたものです。

Python のプログラムを実行すると、それを構成しているそれぞれの文は、原則として、上から下へという順番で 1 回ずつ実行されます。

2.4.3 複数の式文から構成されるプログラムの例

それでは、実際に、複数の式文から構成されるプログラムを書いて、それがどのように実行されるかを試してみましょう。

プログラムの例 sequence.py

```
print('Kaname Madoka')
print('Akemi Homura')
print('Miki Sayaka')
```

このプログラムを、

```
python sequence.py
```

というコマンドで実行すると、次のように文字列が出力されます。

```
Kaname Madoka
Akemi Homura
Miki Sayaka
```

第 3 章 識別子

3.1　識別子の基礎

3.1.1　識別子とは何か

Python のプログラムでは、しばしば、オブジェクトに名前を与えておいて、その名前によってオブジェクトを識別する、ということをします。オブジェクトに名前として与えることのできるものは、「識別子」(identifier) と呼ばれます。

識別子を名前としてオブジェクトに与えることを、識別子をオブジェクトに「束縛する」(bind) と言います。識別子をオブジェクトに束縛することを、識別子にオブジェクトを「代入する」(assign) と言うこともあります。

3.1.2　識別子の作り方

識別子は、次のような規則に従って作ることになっています。

- 識別子を作るために使うことのできる文字は、英字、数字、アンダースコア (_) です。
- 識別子の先頭の文字として、数字を使うことはできません。
- 予約語 (reserved word) と同じものは識別子としては使えません。「予約語」(reserved word) というのは、用途があらかじめ予約されている単語のことで、次のようなものがあります。

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

識別子として使うことのできるものの例としては、次のようなものがあります。

```
a  A  a8  namako  back_to_the_future
```

英字の大文字と小文字は区別されますので、たとえば、a と A は、それぞれを異なるオブジェクトに束縛することができます。

最後の例のように、アンダースコアは、複数の単語から構成される識別子を作るときに、空白の代わりとして使うことができます。

識別子として使うことのできないものの例としては、次のようなものがあります。

`nam@ko`　使うことのできない文字を含んでいる。

`8a`　先頭の文字が数字。

`def`　同じ予約語が存在する。

3.1.3　識別子の値

識別子は、式として評価することができます。識別子を評価すると、その識別子が束縛されているオブジェクトが、その値として得られます。

たとえば、len や print などの組み込み関数の名前は、組み込み関数に束縛されている識別子ですので、評価すると、その値として組み込み関数が得られます。

```
>>> len
<built-in function len>
```

3.2　代入文

3.2.1　代入文の基礎

識別子をオブジェクトに束縛したいとき、言い換えれば、識別子にオブジェクトを代入したいときは、「代入文」(assignment statement) と呼ばれる文を書きます。

代入文は、基本的には、

```
識別子 = 式
```

と書きます。代入文を実行すると、イコールの左側に書かれた識別子が、イコールの右側に書かれた式の値に束縛されます。言い換えれば、イコールの右側に書かれた式の値が、イコールの左側に書かれた識別子に代入されます。たとえば、

```
a = 7
```

という代入文を書くことによって、a という識別子が 7 という整数に束縛されます。言い換えれば、a という識別子に 7 という整数が代入されます。

```
>>> a = 7
>>> a
7
```

3.2.2 再束縛

すでに何らかのオブジェクトに束縛されている識別子について、その束縛を解消して、そのオブジェクトとは別のオブジェクトに束縛することを、識別子をオブジェクトに「再束縛する」(rebind) と言います。

代入文のイコールの左側に、すでに何らかのオブジェクトに束縛されている識別子を書いたとすると、その識別子は、イコールの右側の式の値に再束縛されます。

```
>>> a = 7
>>> a
7
>>> a = 3
>>> a
3
```

3.2.3 累算代入文

代入文には、「累算代入文」(augmented assignment statement) と呼ばれる変種があります。

累算代入文は、

識別子 *op*= 式

と書きます。この中の *op* というところには、何らかの二項演算子を書きます。

累算代入文は、すでに何らかのオブジェクトに束縛されている識別子を、そのオブジェクトに対して二項演算子の動作を実行した結果に再束縛する、という動作をします。たとえば、a という識別子が 7 という整数に束縛されているとするとき、

```
a *= 3
```

という累算代入文を実行すると、a は、7 を 3 倍した結果、すなわち 21 に再束縛されます。

```
>>> a = 7
>>> a *= 3
>>> a
21
```

3.3 関数定義

3.3.1 関数定義の基礎

関数を生成して、識別子をその関数に束縛することを、関数を「定義する」(define) と言います。

関数は、「関数定義」(function definition) と呼ばれる文を書くことによって定義することができます。関数定義は、関数を定義するという動作を意味する文です。

3.3.2 基本的な関数定義

引数を受け取らない関数を定義する関数定義は、

```
def 識別子 (): 文
```

と書きます。関数定義を実行すると、「文」のところに書かれた文があらわしている動作を実行する関数が生成されて、「識別子」のところに書かれた識別子がその関数に束縛されます。

たとえば、次のような関数定義を書くことによって、「Hello, world!」という文字列を出力する関数を生成して、hello という識別子をその関数に束縛することができます。

```
def hello(): print('Hello, world!')
```

インタラクティブシェルに関数定義を入力すると、インタラクティブシェルは、

```
...
```

というプロンプトを表示して続きの入力を要求しますが、エンターキーを押すことによって、「入力はこれで終わりです」ということをインタラクティブシェルに知らせることができます。

```
>>> def hello(): print('Hello, world!')
...
>>> hello()
Hello, world!
```

次に、関数定義をファイルに保存しておいて、インタラクティブシェルでその関数定義を実行してみましょう。

プログラムの例 funcdef.py

```
def world(): print('What a wonderful world!')
```

実行例

```
>>> from funcdef import *
>>> world()
What a wonderful world!
```

3.3.3 複数の文から構成される関数定義

関数の動作を記述するための文は、1 個だけではなくて、何個でも書くことができます。関数の動作を複数の文で記述したいときは、「ブロック」(block) と呼ばれるものを書きます。

ブロックというのは、改行で区切って文を並べたもののことです。ただし、それらの文は、インデントする必要があります。文を「インデントする」(indent) というのは、その先頭に 1 個以上の空白またはタブを書くということです。ひとつのブロックを構成しているそれぞれの文は、同じ個数の空白またはタブによってインデントされている必要があります。Python では、インデントには 4 個の空白を使うことが推奨されています。

ブロックは、それを構成しているそれぞれの文を、原則として、上から下へという順番で 1 回ずつ実行する、という動作をあらわしています。

関数の動作を複数の文で記述したいときは、

```
def 識別子 ():
    ブロック
```

という形の関数定義を書きます。この形の関数定義を実行すると、ブロックがあらわしている動作を実行する関数が生成されます。

```
>>> def namako():
...     print('namako')
...     print('umiushi')
...     print('kurage')
...
>>> namako()
namako
umiushi
kurage
```

プログラムの例 magica.py

```
def magica():
    print('Kaname Madoka')
    print('Akemi Homura')
    print('Miki Sayaka')
```

実行例

```
>>> from magica import *
>>> magica()
Kaname Madoka
Akemi Homura
Miki Sayaka
```

3.4 スコープ

3.4.1 スコープの基礎

識別子とオブジェクトとの束縛が有効である、プログラムの上での範囲は、その識別子の「スコープ」(scope) と呼ばれます。

3.4.2 モジュールスコープ

Python では、関数定義の外でオブジェクトに束縛された識別子は、ひとつのファイルの中に保存されたプログラムの全域というスコープを持つことになります。そのようなスコープは、「モジュールスコープ」(module scope) と呼ばれます。

関数定義の内部もモジュールスコープの一部分ですから、関数定義の中でモジュールスコープを持つ識別子を評価すると、その識別子が束縛されているオブジェクトが値として得られます。

プログラムの例 module.py

```
def func():
    print('func:', a)

a = 'I am a string.'
func()
```

実行例

```
func: I am a string.
```

3.4.3 ローカルスコープ

Python では、関数定義の中でオブジェクトに束縛された識別子は、その関数定義の中だけというスコープを持つことになります。そのような、関数定義の中だけという限定されたスコープは、「ローカルスコープ」(local scope) と呼ばれます。

モジュールスコープを持つ識別子と同一の識別子を、ローカルスコープを持つ識別子として使っても、問題はありません。

プログラムの例 local.py

```
def func():
    a = 'My scope is func.'
    print(a)
    funcfunc()
    print(a)

def funcfunc():
    a = 'My scope is funcfunc.'
    print(a)

a = 'My scope is module.'
print(a)
func()
print(a)
```

実行例

```
My scope is module.
My scope is func.
My scope is funcfunc.
My scope is func.
My scope is module.
```

3.4.4 ローカルスコープのメリット

ところで、「関数定義の中でオブジェクトに束縛された識別子は、その関数定義の中だけというスコープを持つ」という規則には、いったいどのようなメリットがあるのでしょうか。

もしも、「ひとつの関数定義の中で束縛された識別子は、それとは別の関数定義の中でも有効である」という規則が定められていたとするとどうなるか、ということについて考えてみましょう。その場合、識別子をオブジェクトに束縛するときには、その識別子がすでに別の関数定義で

使われていないか、ということに細心の注意を払う必要があります。うっかりと同一の識別子を複数の関数定義の中で使うと、思わぬ不具合が発生しかねません。

つまり、「関数定義の中でオブジェクトに束縛された識別子は、その関数定義の中だけというスコープを持つ」という規則は、「関数定義を書くときに、その関数定義の外でどのような識別子が使われているかということを、まったく気にする必要がない」というメリットを、プログラムを書く人に与えてくれているのです。

3.4.5 global 文

関数定義の中で識別子をオブジェクトに束縛すると、その識別子はローカルスコープを持つことになります。それでは、モジュールスコープを持つ識別子を関数定義の中で再束縛したい、というときにはどうすればいいのでしょうか。

そのような場合に必要となるのが、「global 文」(global statement) と呼ばれる文です。この文を関数定義の中に書いておけば、モジュールスコープを持つ識別子を関数定義の中で再束縛することができるようになります。

global 文は、

```
global 識別子, ···
```

と書きます。この文は、「識別子」のところに書かれた識別子を、モジュールスコープを持つものとして認識する、という動作をあらわしています。たとえば、関数定義の中に、

```
global a
```

という global 文を書いておくと、モジュールスコープを持つ a という識別子を、その関数定義の中で別のオブジェクトに再束縛することができるようになります。

プログラムの例 global.py

```
def func():
    global a
    print('func:', a)
    a = 'I am another string.'
    print('func:', a)

a = 'I am a string.'
print(a)
func()
print(a)
```

実行例

```
I am a string.
func: I am a string.
func: I am another string.
I am another string.
```

3.5 引数

3.5.1 仮引数

引数を受け取る関数を定義する関数定義は、

```
def 識別子 (識別子, ··· ): 文
```

と書くか、または、

```
def 識別子 (識別子, ··· ):
    ブロック
```

と書きます。つまり、丸括弧の中に何個かの識別子を書くわけです。丸括弧の中に書かれた識別子は、「仮引数」(formal argument) と呼ばれます。仮引数は、関数が呼び出されたとき、関数が受け取った引数に束縛されることになります。

仮引数は、ローカルスコープを持つことになります。つまり、仮引数のスコープは関数定義の中だけです。

次のプログラムの中で定義されている stars という関数は、引数として受け取った個数のアスタリスクを出力します。

プログラムの例 stars.py

```
def stars(length): print('*' * length)
```

実行例

```
>>> from stars import *
>>> stars(40)
****************************************
```

3.5.2 仮引数の順序

関数定義の中で並んでいるそれぞれの仮引数と、呼び出しの中で並んでいるそれぞれの式とは、同じ順序で結び付けられます。たとえば、

```
def namako(a, b, c): 文
```

という関数定義で定義された関数を、

```
namako(24, 33, 81)
```

という呼び出しで呼び出したとすると、 a が 24 に、 b が 33 に、 c が 81 に束縛されることになります。

次のプログラムの中で定義されている rect という関数は、文字列と横の個数と縦の個数を引数として受け取って、その文字列を長方形の形に並べたものを出力します。

プログラムの例 rect.py

```
def rect(str, width, height):
    print((str * width + '\n') * height, end='')
```

実行例

```
>>> from rect import *
>>> rect('Python', 4, 3)
PythonPythonPythonPython
PythonPythonPythonPython
PythonPythonPythonPython
```

print は、デフォルトでは 1 個の改行を最後に出力します。この改行を抑制したい場合は、rect の定義の中に書かれているように、 end というキーワードで、長さが 0 の文字列を渡します。ちなみに、長さが 0 の文字列は、「空文字列」(empty string) と呼ばれます。

3.5.3 デフォルト値

仮引数は、関数を定義する段階で、あらかじめ特定のオブジェクトに束縛しておくことができます。仮引数があらかじめ束縛されているオブジェクトは、「デフォルト値」(default value) と呼ばれます。仮引数をあらかじめデフォルト値に束縛しておくと、その仮引数に引数が渡されなかった場合、その仮引数はデフォルト値に束縛されたままになりますので、引数の代わりとしてデフォルト値が使われることになります。

仮引数をデフォルト値に束縛したいときは、関数定義の先頭の丸括弧の中に、

```
仮引数 = 式
```

という形のものを書きます。そうすると、その中に書かれた式の値が、仮引数のデフォルト値になります。

プログラムの例 rect2.py

```
def rect2(str='Python', width=4, height=3):
    print((str * width + '\n') * height, end='')
```

rect2 という関数をこのように定義したとすると、引数を 2 個しか渡さなかった場合には height がデフォルト値のままになり、引数を 1 個しか渡さなかった場合には width と height がデフォルト値のままになり、引数をまったく渡さなかった場合には str と width と height がデフォルト値のままになります。

実行例

```
>>> from rect2 import *
>>> rect2('Ruby', 7, 2)
RubyRubyRubyRubyRubyRubyRuby
RubyRubyRubyRubyRubyRubyRuby
>>> rect2('Ruby', 7)
RubyRubyRubyRubyRubyRubyRuby
RubyRubyRubyRubyRubyRubyRuby
RubyRubyRubyRubyRubyRubyRuby
>>> rect2('Ruby')
RubyRubyRubyRuby
RubyRubyRubyRuby
RubyRubyRubyRuby
>>> rect2()
PythonPythonPythonPython
PythonPythonPythonPython
PythonPythonPythonPython
```

3.5.4 キーワードとしての仮引数

第 2.3.9 項で説明したように、関数には、「キーワード引数」(keyword argument) と呼ばれるものを渡すことができます。

仮引数は、キーワード引数を関数に渡すときに使われるキーワードになります。

それでは、先ほど定義を書いた `rect2` に、キーワード引数を渡してみましょう。

実行例

```
>>> rect2(height=2, str='Haskell', width=6)
HaskellHaskellHaskellHaskellHaskellHaskell
HaskellHaskellHaskellHaskellHaskellHaskell
```

キーワード引数を使えば、順番が前の仮引数には引数を渡さないで、後ろの仮引数に引数を渡す、ということもできます。

実行例

```
>>> rect2(height=4)
PythonPythonPythonPython
PythonPythonPythonPython
PythonPythonPythonPython
PythonPythonPythonPython
```

3.6 戻り値

3.6.1 return 文

戻り値を返す関数を定義したいときは、「`return` 文」(`return` statement) と呼ばれる文を関数定義の中に書きます。

`return` 文は、

```
return 式
```

と書きます。この中の「式」というところには、戻り値として返したいオブジェクトを求める式を書きます。`return` 文は、式を評価して、その値を戻り値にして、関数の動作を終了させる、という動作をあらわしています。

a が 0 でない数値だとするとき、$1/a$ という数値は、a の「逆数」(reciprocal, multiplicative inverse) と呼ばれます。次のプログラムの中で定義されている `recipro` という関数は、1 個の数値を引数として受け取って、その逆数を戻り値として返します。

プログラムの例 `recipro.py`

```
def recipro(a): return 1/a
```

実行例

```
>>> from recipro import *
>>> recipro(4)
```

```
0.25
```

n が自然数だとするとき、1 から n までの自然数の総和は、

$$\frac{n(n+1)}{2}$$

という計算をすることによって求めることができます。次のプログラムの中で定義されている sum という関数は、1 個の自然数を引数として受け取って、1 からその自然数までの総和を戻り値として返します。

プログラムの例　sum.py

```
def sum(n): return n*(n+1)//2
```

実行例

```
>>> from sum import *
>>> sum(10)
55
```

3.6.2　return 文を実行しないで終了した関数の戻り値

Python では、あらゆる関数が戻り値を返します。関数が return 文を実行しないで動作を終了した場合も、その関数は戻り値を返しています。その場合に関数が返すのは、None と呼ばれる特定のオブジェクトです。

None は、None という予約語を評価したときに値として得られるオブジェクトです。

None のような、評価すると特定のオブジェクトが値として得られる予約語は、「組み込み定数」(built-in constant) と呼ばれます。

第 4 章　選択

4.1　選択の基礎

4.1.1　選択とは何か

第 2.4.2 項で説明したように、Python のプログラムは、いくつかの文を改行で区切って並べたものです。プログラムを構成しているそれぞれの文は、原則として、上から下へという順番で 1 回ずつ実行されます。

ですから、いくつかの文を書くことによって、まずこの動作を実行して、次にこの動作を実行して、次にこの動作を実行して……というように、複数の動作を直線的に実行していくという動作を記述することができるわけですが、コンピュータに実行させたい動作は、必ずしも一直線に進んでいくものばかりとは限りません。しばしば、そのときの状況に応じて、いくつかの動作の候補の中からひとつの動作を選んで実行するということも、必要になります。

「いくつかの動作の候補の中からひとつの動作を選んで実行する」という動作は、「選択」(selection) と呼ばれます。

4.1.2　条件

コンピュータは、運を天に任せて動作を選択するわけではありません。選択は、何らかの判断にもとづいて実行されます。

成り立っているか、それとも成り立っていないか、という判断の対象は、「条件」(condition) と呼ばれます。

条件が成り立っていると判断されるとき、その条件は「真」(true) であると言われます。逆に、条件が成り立っていないと判断されるとき、その条件は「偽」(false) であると言われます。

4.1.3　真偽値

真を意味するオブジェクトと、偽を意味するオブジェクトは、総称して「真偽値」(Boolean) と呼ばれます。

何の判断もしないで、特定の真偽値を求めたいときは、次の二つの組み込み定数のうちのどちらかを書きます。

`True`　真を値とする組み込み定数
`False`　偽を値とする組み込み定数

4.2　比較演算子

4.2.1　比較演算子の基礎

二つのデータのあいだに何らかの関係があるという条件が成り立っているかどうかを調べる二項演算子は、「比較演算子」(comparison operator) と呼ばれます。

比較演算子の優先順位は、加算や乗算などの演算子よりも低くなっています。

比較演算子を含む式を評価すると、演算子の左右にある式が評価されて、それらの式の値のあいだに関係が成り立っているかどうかという判断が実行されます。そして、関係が成り立っているならば真、成り立っていないならば偽が、式全体の値になります。

4.2.2　大小関係

次の比較演算子を使うことによって、オブジェクトの大小関係について調べることができます。

a `>` b　a は b よりも大きい。
a `<` b　a は b よりも小さい。
a `>=` b　a は b よりも大きいか、または a と b とは等しい。
a `<=` b　a は b よりも小さいか、または a と b とは等しい。

```
>>> 8 > 5
True
>>> 5 > 8
False
>>> 5 > 5
False
>>> 5 >= 5
True
```

大小関係があるのは、数値と数値とのあいだだけではありません。文字列と文字列とのあいだにも大小関係があります。

辞書の見出しは、「辞書式順序」(lexicographical order) と呼ばれる順序で並べられています。文字列と文字列とのあいだの大小関係は、辞書式順序で文字列を並べたときに、後ろにあるものは前にあるものよりも大きい、という関係です。

```
>>> 'stay' > 'star'
True
>>> 'star' > 'stay'
False
```

4.2.3　等しいかどうか

二つのオブジェクトが等しいかどうかということは、次の比較演算子を使うことによって調べることができます。

a `==` b　a と b とは等しい。
a `!=` b　a と b とは等しくない。

```
>>> 5 == 5
True
>>> 5 == 8
False
>>> 'star' == 'star'
True
>>> 'star' == 'stay'
False
```

4.3 if 文

4.3.1 if 文の基礎

何らかの条件が成り立っているかどうかを調べて、その結果にもとづいて二つの動作のうちのどちらかを実行したい、というときは、「if 文」(if statement) と呼ばれる文を書きます。

if 文は、

```
if 条件式:
    ブロック 1
else:
    ブロック 2
```

と書きます。この中の「条件式」のところには、条件をあらわす式、つまり、評価すると値として真偽値が得られる式を書きます。

if 文を実行すると、まず最初に、条件式が評価されます。そして、条件式の値が真だった場合は、ブロック $_1$ が実行されます（その場合、ブロック $_2$ は実行されません)。条件式の値が偽だった場合は、ブロック $_2$ が実行されます（その場合、ブロック $_1$ は実行されません)。

つまり、if 文は、「もしも条件式が真ならばブロック $_1$ を実行して、そうでなければブロック $_2$ を実行する」という動作を意味しているわけです。

それでは、if 文をインタラクティブシェルに入力してみましょう。

```
>>> if 8 > 5:
...     print('namako')
... else:
...     print('hitode')
...
namako
```

この場合、条件式の値は真ですので、namako が出力されて、hitode は出力されません。

```
>>> if 5 > 8:
...     print('namako')
... else:
...     print('hitode')
...
hitode
```

この場合、条件式の値は偽ですので、hitode が出力されて、namako は出力されません。

次のプログラムの中で定義されている evenodd という関数は、整数を受け取って、それが偶数ならば「偶数」という文字列を返して、そうでなければ「奇数」という文字列を返します。

プログラムの例 evenodd.py

```
def evenodd(n):
    if n%2 == 0:
        return '偶数'
    else:
        return '奇数'
```

実行例

```
>>> from evenodd import *
>>> evenodd(6)
'偶数'
>>> evenodd(7)
'奇数'
```

4.3.2 else 以降を省略した if 文

二つの動作のうちのどちらかを選択するのではなくて、ひとつの動作を実行するかしないかを選択したい、ということもしばしばあります。そのような場合は、else 以降を省略した if 文を書きます。つまり、

```
if 条件式:
    ブロック
```

という形の if 文を書くわけです。この形の if 文を実行すると、条件式の値が真の場合はブロックが実行されますが、条件式の値が偽の場合は何も実行されません。

インタラクティブシェルを使って試してみましょう。

```
>>> if 8 > 5:
...     print('namako')
...
namako
>>> if 5 > 8:
...     print('namako')
...
>>>
```

次のプログラムの中で定義されている mtohm という関数は、分を単位とする時間の長さを受け取って、それを「何時間何分」という形式の文字列に変換した結果を返します。ただし、「何分」という端数が出ない場合は、「何時間」という形式の文字列を返します。

プログラムの例　mtohm.py

```
def mtohm(m):
    hm = str(m//60) + '時間'
    if m%60 != 0:
        hm += str(m%60) + '分'
    return hm
```

実行例

```
>>> from mtohm import *
>>> mtohm(160)
'2時間40分'
>>> mtohm(180)
'3時間'
```

「何時間」の部分の右側に「何分」の部分を連結するという動作は、実行するかしないかを選択することになりますので、このように、else 以降を省略した if 文を使って書くことができます。

4.3.3　多肢選択

選択の対象となる動作が 3 個以上あるような選択は、「多肢選択」(multibranch selection) と呼ばれます。

多肢選択を記述したいときは、 if 文の中に、

```
elif 条件式:
    ブロック
```

という形のものをいくつか書きます。この中の elif というのは、else if を縮めたもので、「そうではなくてもしも……ならば」ということを意味しています。

たとえば、3 個の条件による多肢選択は、

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

という形の if 文を書くことによって記述することができます。この形の if 文を実行すると、値が真になる条件式が見つかるまで、条件式 $_1$、条件式 $_2$、条件式 $_3$ という順番で条件式が評価されていきます。値が真になる条件式が見つかった場合は、その条件式と同じ番号のブロックが実行されます（その場合、それ以降の条件式は評価されません）。すべての条件式の値が偽だった場合は、ブロック $_4$ が実行されます。

インタラクティブシェルを使って試してみましょう。

```
>>> if 5 > 8:
...     print('namako')
... elif 8 > 5:
...     print('hitode')
... else:
...     print('umiushi')
...
hitode
```

次のプログラムの中で定義されている sign という関数は、数値を受け取って、それがプラスならば「プラス」という文字列を返して、そうでなくてマイナスならば「マイナス」という文字列を返して、そうでなければ「ゼロ」という文字列を返します。

プログラムの例 sign.py

```
def sign(a):
    if a > 0:
        return 'プラス'
    elif a < 0:
        return 'マイナス'
    else:
        return 'ゼロ'
```

実行例

```
>>> from sign import *
>>> sign(5)
'プラス'
>>> sign(-5)
'マイナス'
>>> sign(0)
'ゼロ'
```

4.3.4 複合文

文を組み合わせることによって作られる文は、「複合文」(compound statement) と呼ばれます。 if 文は、複合文の一種です。

複合文ではない文、つまり、これ以上は文に分解することのできない文は、「単純文」(simple statement) と呼ばれます。式文、代入文、 return 文などは単純文に分類されます。

1 個の単純文だけを実行する関数を定義する関数定義は、たとえば、

```
def sum(n): return n*(n+1)//2
```

というように、ブロックを使わなくても書くことができます。しかし、複合文を実行する関数を定義する関数定義は、たとえ 1 個の複合文を実行するだけであっても、ブロックを使う必要があります。

4.4 論理演算子

4.4.1 論理演算子の基礎

処理の対象が真偽値で、処理の結果も真偽値であるような動作をあらわしている演算子は、「論理演算子」(Boolean operator) と呼ばれます。

Python には、次の三つの論理演算子があります。

a and b　　a かつ b である。

a or b　　a または b である。

not a　　a ではない。

and と or は、比較演算子よりも低い優先順位を持っています。そして、 and は、 or よりも高い優先順位を持っています。

4.4.2 論理積演算子

and は、「論理積演算子」(Boolean AND operator) と呼ばれます。これは、二つの条件が両方とも成り立っているかどうかを判断したいとき、つまり、A かつ B という条件が成り立ってい

るかどうかを判断したいときに使われる論理演算子で、次のような動作をします。

```
True   and  True     ⟶  True
True   and  False    ⟶  False
False  and  True     ⟶  False
False  and  False    ⟶  False
```

4.4.3　論理和演算子

or は、「論理和演算子」(Boolean OR operator) と呼ばれます。これは、二つの条件のうちの少なくとも一つが成り立っているかどうかを判断したいとき、つまり、A または B という条件が成り立っているかどうかを判断したいときに使われる論理演算子で、次のような動作をします。

```
True   or  True     ⟶  True
True   or  False    ⟶  True
False  or  True     ⟶  True
False  or  False    ⟶  False
```

次のプログラムの中で定義されている leapyear という関数は、西暦であらわされた年を受け取って、その年がうるう年ならば「うるう年」という文字列を返して、そうでなければ「平年」という文字列を返します。

プログラムの例　leapyear.py

```
def leapyear(y):
    if y%4 == 0 and y%100 != 0 or y%400 == 0:
        return ’うるう年’
    else:
        return ’平年’
```

実行例

```
>>> from leapyear import *
>>> leapyear(2080)
’うるう年’
>>> leapyear(2100)
’平年’
>>> leapyear(2400)
’うるう年’
```

4.4.4　論理否定演算子

not は、「論理否定演算子」(Boolean negation operator) と呼ばれます。これは、真偽値を反転させたいとき、つまり、A ではないという条件が成り立っているかどうかを判断したいときに使われる論理演算子で、次のような動作をします。

```
not True     ⟶  False
not False    ⟶  True
```

4.5　条件演算式

4.5.1　条件演算式の基礎

第 4.3 節で説明したように、Python では、if 文という文を使うことによって選択を記述することができます。しかし、選択を記述する方法は、それだけではありません。

Python では、選択は、「条件演算式」(conditional expression) と呼ばれる式を書くことによって記述することも可能です。

4.5.2　条件演算式の書き方

条件演算式は、

式 $_1$ if 条件式 else 式 $_2$

と書きます。「条件式」のところには、評価すると値として真偽値が得られる式を書きます。

条件演算式を評価すると、まず最初に、条件式が評価されます。そして、条件式の値が真だった場合は、式$_1$が評価されます（その場合、式$_2$は評価されません）。条件式の値が偽だった場合は、式$_2$が評価されます（その場合、式$_1$は評価されません）。そして、評価された式$_1$または式$_2$の値が、条件演算式の値になります。

インタラクティブシェルを使って試してみましょう。

```
>>> 'namako' if 8 > 5 else 'hitode'
'namako'
>>> 'namako' if 5 > 8 else 'hitode'
'hitode'
```

次のプログラムは、第 4.3.1 項で紹介した、整数が偶数なのか奇数なのかを判定するプログラムを、条件演算式を使って書き直したものです。

プログラムの例　evenodd2.py

```
def evenodd(n): return '偶数' if n%2 == 0 else '奇数'
```

実行例

```
>>> from evenodd2 import *
>>> evenodd(6)
'偶数'
>>> evenodd(7)
'奇数'
```

第 5 章　繰り返しと再帰

5.1　繰り返しの基礎

5.1.1　繰り返しとは何か

コンピュータに実行させたい動作は、必ずしも、一連の動作をそれぞれ一回ずつ実行していけばそれで達成される、というものばかりとは限りません。しばしば、ほとんど同じ動作を何回も何十回も何百回も実行しなければ意図していることを達成できない、ということがあります。

「同じ動作を何回も実行する」という動作は、「繰り返し」(iteration) と呼ばれます。

この章では、繰り返しというのはどのように記述すればいいのか、ということについて説明します。

5.1.2　繰り返しを記述するための文

繰り返しは、繰り返したい回数と同じ個数の文を書くことによって記述することも可能ですが、そのような書き方だと、繰り返しの回数に比例してプログラムが長くなってしまいます。ですから、多くのプログラミング言語は、繰り返しを簡潔に記述することができるようにする機能を持っています。

Python では、次の 2 種類の文のうちのどちらかを使うことによって、繰り返しを簡潔に記述することができます。

- `for` 文 (`for` statement)
- `while` 文 (`while` statement)

5.2　`for` 文

5.2.1　`for` 文の基礎

Python では、それに対して繰り返しを実行することが可能なオブジェクトのことを、「繰り返し可能オブジェクト」(iterable object) と呼びます。

たとえば、文字列は、それを構成しているそれぞれの文字に対して繰り返しを実行することのできるオブジェクトですので、繰り返し可能オブジェクトの一種です。

繰り返し可能オブジェクトに対して繰り返しを実行したいときは、通常、「for 文」(for statement) と呼ばれる文が使われます。

5.2.2　for 文の書き方

for 文は、

```
for 識別子 in 式:
    ブロック
```

と書きます。この中の「式」のところには、値として繰り返し可能オブジェクトが得られる式を書きます。

for 文を実行すると、まず最初に、in の右側の式が 1 回だけ評価されます。そして、その式の値として得られた繰り返し可能オブジェクトに対して、ブロックの実行が繰り返されます。for の右側の識別子は、ブロックが実行される直前に、ブロックの動作の対象となるオブジェクトに束縛されます。たとえば、

```
for c in 'namako':
    print(c, end=', ')
```

という for 文を実行すると、まず最初に、'namako' という文字列リテラルが評価されて、namako という文字列が値として得られます。そして、その文字列を構成している 1 個 1 個の文字に対して、

```
print(c, end=', ')
```

というブロックが実行されます（ここで「文字」と呼んでいるのは、厳密に言えば、1 文字だけから構成される文字列のことです)。c という識別子は、ブロックが実行される直前に、ブロックの動作の対象となる文字に束縛されます。

インタラクティブシェルを使って試してみましょう。

```
>>> for c in 'namako':
...     print(c, end=', ')
...
n, a, m, a, k, o, >>>
```

次のプログラムの中で定義されている reverse という関数は、文字列を受け取って、それを構成しているそれぞれの文字を逆の順序で並べ替えることによってできる文字列を返します。

プログラムの例　reverse.py

```
def reverse(s):
    s2 = ''
    for c in s:
        s2 = c + s2
    return s2
```

実行例

```
>>> from reverse import *
>>> reverse('isoginchaku')
'ukahcnigosi'
```

5.2.3　範囲

Python では、「範囲」(range) と呼ばれるオブジェクトを扱うことができます。

範囲というのは、整数から構成される有限の長さの等差数列のことです。つまり、何らかの整数を出発点として、それに対して一定の整数を次々と加算していくことによってできる整数の列のことです。

範囲は、range という組み込み型から生成されるオブジェクトです。

引数として整数 b を渡して range を呼び出すと、range は、

0、1、2、3、・・・、$b-1$

という範囲を生成します。

```
>>> for i in range(10):
...     print(i, end=' ')
```

```
...
0 1 2 3 4 5 6 7 8 9 >>>
```

引数として整数 a、整数 b を渡して range を呼び出すと、 range は、

a、$a+1$、$a+2$、$a+3$、・・・、$b-1$

という範囲を生成します。

```
>>> for i in range(4, 10):
...     print(i, end=' ')
...
4 5 6 7 8 9 >>>
```

引数として整数 a、整数 b、整数 c を渡して range を呼び出すと、 range は、a から始まって、c ずつ増えていって、b に到達する直前で終わる範囲を生成します。

```
>>> for i in range(40, 100, 10):
...     print(i, end=' ')
...
40 50 60 70 80 90 >>>
>>> for i in range(100, 40, -10):
...     print(i, end=' ')
...
100 90 80 70 60 50 >>>
```

次のプログラムの中で定義されている divisor という関数は、プラスの整数を受け取って、そのすべての約数を出力します。

プログラムの例 divisor.py

```
def divisor(n):
    for i in range(1, n+1):
        if n%i == 0:
            print(i, end=' ')
    print()
```

実行例

```
>>> divisor(96)
1 2 3 4 6 8 12 16 24 32 48 96
```

divisor の定義の最後に書かれている、

```
print()
```

という文は、「改行を出力する」という意味です。このように、引数を何も渡さないで print を呼び出した場合、 print は、改行だけを出力します。

5.3 while 文

5.3.1 while 文の基礎

繰り返しを記述したいときは、基本的には for 文を使えばいいわけですが、 for 文では記述することが困難な繰り返しも存在します。たとえば、条件による繰り返しは、 for 文では記述することが困難です。

条件による繰り返しというのは、繰り返しの対象となる動作を実行するたびに、繰り返しを続行するか終了するかということを、何らかの条件が成り立っているかどうかを判断することによって決定する、というタイプの繰り返しのことです。このようなタイプの繰り返しは、 for 文では記述することが困難です。

そこで登場するのが、条件による繰り返しを表現するために存在する、「while 文」(while statement) と呼ばれる文です。

5.3.2 while 文の書き方

while 文は、

```
while 条件式:
    ブロック
```

と書きます。この中の「条件式」のところには、条件をあらわす式、つまり、評価すると値として真偽値が得られる式を書きます。

while 文は、次のような動作をあらわしています。

(1) 条件式を評価する。その値が偽だった場合、 while 文の動作は終了する。
(2) 条件式の値が真だった場合は、ブロックを実行する。
(3) (1) に戻って、ふたたび同じ動作を実行する。

5.3.3 無限ループ

while 文を使うと、永遠に終わらない繰り返しというものを記述することも可能になります。永遠に終わらない繰り返しは、「無限ループ」(infinite loop) と呼ばれます。

True という組み込み定数を評価すると、真を意味するオブジェクトが値として得られますので、 while 文の条件式としてそれを書くと、その while 文は無限ループになります。

たとえば、次の while 文は、 kurage という文字列の出力を永遠に繰り返します。

```
while True:
    print('kurage')
```

この while 文の実行を終了させたいときは、Ctrl-C、つまりコントロールキーを押しながら C のキーを押す、という操作をします。

5.3.4 条件による繰り返しの例

条件による繰り返しの例として、二つの整数の最大公約数を求めるという処理について考えてみることにしましょう。

n がプラスの整数で、m が 0 またはプラスの整数だとするとき、n と m の両方に共通する約数のうちで最大のもののことを、n と m の「最大公約数」(greatest common measure, GCM) と呼びます（m が 0 の場合は、n と m の最大公約数は n だと定義します)。たとえば、54 と 36 の最大公約数は 18 です。

二つの整数の最大公約数は、「ユークリッドの互除法」(Euclidean algorithm) と呼ばれる方法を使えば、きわめて簡単に求めることができます。ユークリッドの互除法というのは、

ステップ 1　与えられた二つの整数のそれぞれを、n と m という変数に代入する。
ステップ 2　m が 0 ならば計算を終了する。
ステップ 3　n を m で除算して、そのあまりを r という変数に代入する。
ステップ 4　m を n に代入する。
ステップ 5　r を m に代入する。
ステップ 6　ステップ 2 に戻る。

という計算を実行していけば、計算が終了したときの n が、最初に与えられた二つの整数の最大公約数になっている、というものです。ステップ 2 からステップ 6 までは、

m が 0 ではないあいだ、ステップ 3 からステップ 5 までを繰り返す。

ということだと考えることができますので、その部分は、 while 文を書くことによって記述することができます。

次のプログラムの中で定義されている gcm という関数は、二つの整数を受け取って、それらの最大公約数を返します。

プログラムの例　gcm.py

```
def gcm(n, m):
    while m != 0:
        r = n%m
        n = m
        m = r
    return n
```

実行例

```
>>> from gcm import *
>>> gcm(54, 36)
18
```

5.4 再帰

5.4.1 再帰とは何か

この節では、「再帰」(recursion) と呼ばれるものについて説明したいと思います。

再帰というのは、全体と同じものが一部分として含まれているという性質のことです。再帰という性質を持っているものは、「再帰的な」(recursive) と形容されます。

ここに、1 台のカメラと 1 台のモニターがあるとします。まず、それらを接続して、カメラで撮影した映像がモニターに映し出されるようにします。そして次に、カメラをモニターの画面に向けます。すると、モニターの画面には、そのモニター自身が映し出されることになります。そして、映し出されたモニターの画面の中には、さらにモニター自身が映し出されています。このときにモニターの画面に映し出されるのは、再帰という性質を持っている映像、つまり再帰的な映像です。

また、先祖と子孫の関係も再帰的です。なぜなら、先祖と子孫との中間にいる人々も、やはり先祖と子孫の関係で結ばれているからです。

5.4.2 基底

再帰という性質を持っているものは、全体と同じものが一部分として含まれているわけですが、その構造は、内部に向かってどこまでも続いている場合もあれば、どこかで終わっている場合もあります。

再帰的な構造がどこかで終わっている場合、その中心には、その内部に再帰的な構造を持っていない何かがあります。そのような、再帰的な構造の中心にあって、その内部に再帰的な構造を持っていないものは、その再帰的な構造の「基底」(basis) と呼ばれます。

先祖と子孫の関係では、親子関係というのが、その再帰的な構造の基底になります。

5.4.3 関数の再帰的な定義

関数は、再帰的に定義することが可能です。関数を再帰的に定義するというのは、定義される当の関数を使って関数を定義するということです。再帰的な構造を持っている概念を取り扱う関数は、再帰的に定義するほうが、再帰的ではない方法で定義するよりもすっきりした記述になります。

関数を再帰的に定義する場合は、それが循環に陥ることを防ぐために、基底について記述した選択肢を作っておくことが必要になります。

5.4.4 階乗

n が 0 またはプラスの整数だとするとき、n から 1 までの整数をすべて乗算した結果、つまり、

$$n \times (n-1) \times (n-2) \times \cdots \times 1$$

という計算の結果は、n の「階乗」(factorial) と呼ばれて、$n!$ と書きあらわされます。ただし、$0!$ は 1 だと定義します。

たとえば、$5!$ は、

$$5 \times 4 \times 3 \times 2 \times 1$$

という計算をすればいいわけですから、120 ということになります。

階乗という概念は、再帰的な構造を持っています。なぜなら、階乗は、

$$\begin{cases} & 0! = 1 \\ n \geq 1 \text{ ならば} & n! = n \times (n-1)! \end{cases}$$

というように再帰的に定義することができるからです。

階乗を求める関数も、再帰的に定義することができます。次のプログラムは、階乗を求める `factorial` という関数を再帰的に定義しています。

プログラムの例 `factorial.py`

```
def factorial(n):
    if n == 0:
        return 1
    elif n >= 1:
        return n * factorial(n-1)
```

実行例

```
>>> from factorial import *
>>> factorial(5)
120
```

5.4.5　フィボナッチ数列

第 0 項と第 1 項が 1 で、第 2 項以降はその直前の 2 項を足し算した結果である、という数列は、「フィボナッチ数列」(Fibonacci sequence) と呼ばれます。フィボナッチ数列の第 0 項から第 12 項までを表にすると、次のようになります。

n	0	1	2	3	4	5	6	7	8	9	10	11	12
第 n 項	1	1	2	3	5	8	13	21	34	55	89	144	233

フィボナッチ数列というのは再帰的な構造を持っている概念ですので、その第 n 項 (F_n) は、

$$\begin{cases} & F_0 = 1 \\ & F_1 = 1 \\ n \geq 2 \text{ ならば} & F_n = F_{n-2} + F_{n-1} \end{cases}$$

というように再帰的に定義することができます。

フィボナッチ数列の第 n 項を求める関数も、再帰的に定義することができます。次のプログラムは、フィボナッチ数列の第 n 項を求める `fibonacci` という関数を再帰的に定義しています。

プログラムの例　`fibonacci.py`

```
def fibonacci(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    elif n >= 2:
        return fibonacci(n-2) + fibonacci(n-1)
```

実行例

```
>>> from fibonacci import *
>>> fibonacci(7)
21
```

5.4.6　最大公約数

第 5.3.4 項で、条件による繰り返しの例として、ユークリッドの互除法を使って二つの整数の最大公約数を求めるという処理について説明しましたが、ユークリッドの互除法は、二つの整数を n と m とするとき、次のように再帰的に記述することも可能です。

- m が 0 ならば、n が、n と m の最大公約数である。
- m が 1 以上ならば、n を m で除算したときのあまりを求めて、その結果を r とする。そして、m と r の最大公約数を求めれば、その結果が n と m の最大公約数である。

次のプログラムは、2 個のプラスの整数の最大公約数を求める `gcm` という関数を、ユークリッドの互除法を使って定義しています。

プログラムの例　`gcm2.py`

```
def gcm(n, m):
    if m == 0:
        return n
    elif m >= 1:
        return gcm(m, n%m)
```

実行例

```
>>> from gcm2 import *
>>> gcm(54, 36)
18
```

第 6 章　シーケンス

6.1　シーケンスの基礎

6.1.1　スカラーとコレクション

Python によって扱われるオブジェクトは、次の 2 種類に分類することができます。

- スカラー (scalar)
- コレクション (collection)

スカラーというのは、それ以上はオブジェクトに分解することのできないオブジェクトのことです。整数、浮動小数点数、関数、真偽値、None などは、スカラーに分類されるオブジェクトです。

コレクションというのは、いくつかのオブジェクトから構成されているオブジェクトのことです。範囲や文字列は、コレクションに分類されるオブジェクトです。

すべてのコレクションは繰り返し可能オブジェクトですので、`for` 文を使うことによって、それを構成しているそれぞれのオブジェクトについて何らかの動作を繰り返すことができます。

6.1.2　変更不可能オブジェクトと変更可能オブジェクト

オブジェクトは、次の 2 種類に分類することができます。

- 変更不可能オブジェクト (immutable object)
- 変更可能オブジェクト (mutable object)

これらの 2 種類の相違点は、その名前のとおり、変更が可能かどうかです。変更不可能オブジェクトは、ひとたび生成されたのちは、それに対する変更ができません。それに対して、変更可能オブジェクトは、生成されたのちも、それに対する変更が可能です。

変更不可能オブジェクトには、次のようなものがあります。

- スカラー (scalar)
- 文字列 (string)
- 範囲 (range)
- タプル (tuple)
- 不変集合 (frozen set)

変更可能オブジェクトには、次のようなものがあります。

- リスト (list)
- 集合 (set)
- 辞書 (dictionary)

6.1.3　要素

コレクションを構成しているそれぞれのオブジェクトは、そのコレクションの「要素」(item) と呼ばれます。

文字列の要素は、1 個の文字から構成される文字列です。そして、範囲の要素は、整数です。

6.1.4　コレクションの長さ

コレクションが持っている要素の個数は、そのコレクションの「長さ」(length) と呼ばれます。

`len` という関数は、引数としてコレクションを受け取って、戻り値としてその長さを返します。

```
>>> len('isoginchaku')
11
```

```
>>> len(range(30, 40))
10
```

6.1.5　帰属演算子

特定の要素がコレクションに帰属しているかどうかという関係、つまり、コレクションが特定の要素を持っているという関係は、「帰属関係」(membership relation) と呼ばれます。

オブジェクトとコレクションとのあいだに帰属関係があるかどうかということは、「帰属演算子」(membership operator) と呼ばれる比較演算子を使うことによって調べることができます。

帰属演算子には、次の二つのものがあります。

x `in` a　　x は a の要素である。

x `not in` a　　x は a の要素ではない。

通常のコレクションの場合、帰属演算子が調べるのは、オブジェクトが 1 個の要素としてコレクションに帰属しているかどうかという関係です。しかし、文字列の場合、帰属演算子は、文字列の一部分として文字列が含まれているかどうかという関係を調べます。

```
>>> 'inch' in 'isoginchaku'
True
>>> 'bach' in 'isoginchaku'
False
>>> 'inch' not in 'isoginchaku'
False
>>> 'bach' not in 'isoginchaku'
True
>>> 36 in range(30, 40)
True
>>> 46 in range(30, 40)
False
>>> 36 not in range(30, 40)
False
>>> 46 not in range(30, 40)
True
```

6.1.6　最大値と最小値

`max` という関数は、引数としてコレクションを受け取って、そのコレクションを構成している要素のうちでもっとも大きいものを返します。

同じように `min` という関数は、引数としてコレクションを受け取って、そのコレクションを構成している要素のうちでもっとも小さいものを返します。

```
>>> max('isoginchaku')
'u'
>>> min('isoginchaku')
'a'
>>> max(range(30, 40))
39
>>> min(range(30, 40))
30
```

6.1.7　シーケンスとは何か

コレクションは、順序を持っているものと、順序を持っていないものに分類することができます。順序を持っているコレクションは、「シーケンス」(sequence) と呼ばれます。

シーケンスには、次のようなものがあります。

- 文字列 (string)
- 範囲 (range)
- タプル (tuple)
- リスト (list)

タプルについては第 6.2 節で、リストについては第 6.3 節で説明することにしたいと思います。そして、順序を持っていないコレクションについては第 7 章で説明することにしたいと思います。

6.1.8 インデックス

シーケンスを構成しているそれぞれの要素は、「インデックス」(index) と呼ばれる番号によって識別することができます。

インデックスには、2 種類のものがあります。ひとつは、それぞれの要素に対して、先頭から順番に、

0、1、2、3、4、・・・

と与えられるインデックスで、もうひとつは、末尾から順番に、

−1、−2、−3、−4、・・・

と与えられるインデックスです。

6.1.9 添字表記

シーケンスが持っている特定の要素を指定したいときは、「添字表記」(subscription) と呼ばれるものを書きます。

添字表記は、

s[i]

と書きます。s はシーケンスを求める式で、i は要素のインデックスを求める式です。

添字表記は、式として評価することができます。添字表記を式として評価すると、その値として、インデックスで指定されたシーケンスの要素が得られます。

```
>>> 'isoginchaku'[3]
'g'
>>> 'isoginchaku'[-3]
'a'
>>> range(30, 40)[3]
33
>>> range(30, 40)[-3]
37
```

6.1.10 スライス表記

シーケンスの一部分となっているシーケンスは、「部分シーケンス」(subsequence) と呼ばれます。

シーケンスが持っている特定の部分シーケンスを指定したいときは、「スライス表記」(slicing) と呼ばれるものを書きます。

スライス表記は、

s[i:j]

と書きます。s はシーケンスを求める式で、i と j のそれぞれは要素のインデックスを求める式です。指定される部分シーケンスは、i 番目から $j-1$ 番目までです。

スライス表記は、式として評価することができます。スライス表記を式として評価すると、その値として、二つのインデックスで指定された範囲の部分シーケンスが得られます。

```
>>> 'isoginchaku'[4:8]
'inch'
>>> range(30, 40)[4:8]
range(34, 38)
```

コロンの左右の式は、省略することができます。左側の式を省略すると、シーケンスの先頭から始まる部分シーケンスが指定されて、右側の式を省略すると、シーケンスの末尾で終わる部分シーケンスが指定されます。

```
>>> 'isoginchaku'[:6]
'isogin'
>>> range(30, 40)[4:]
range(34, 40)
```

6.1.11 シーケンスの連結

第 2.2.4 項で、+ という演算子を使うことによって文字列の連結ができるという説明をしましたが、文字列だけではなくて、それ以外のシーケンスについても、この演算子を使うことによっ

て、連結を実行することができます（ただし、範囲の連結はできません）。
同じように、* という演算子を使うことによって、文字列だけではなくて、それ以外のシーケンス（ただし範囲を除く）についても、指定された回数だけ同じシーケンスを連結することができます。

6.1.12　要素の探索

すべてのシーケンスは、index というメソッドを持っています。これは、自身の中で、先頭から末尾へ向かって引数を探索して、引数と一致する要素を発見した場合は、その要素のインデックスを戻り値として返す、という動作をするメソッドです。
文字列が持っている index は、引数として文字列を受け取って、引数と一致する部分シーケンスの先頭のインデックスを戻り値として返します。

```
>>> 'abc785def785ghi'.index('785')
3
>>> range(30, 40).index(36)
6
```

引数を発見することができなかった場合は、エラーになります。

```
>>> range(30, 40).index(48)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 48 is not in range
```

2 個目の引数として整数 i を渡して index を呼び出すと、index は i 番目の要素から探索を開始します。

```
>>> 'abc785def785ghi'.index('785', 4)
9
```

2 個目の引数として整数 i、3 個目の引数として整数 j を渡して index を呼び出すと、index は i 番目から $j-1$ 番目までの範囲で探索を実行します。

```
>>> 'abc785def785ghi'.index('785', 4, 11)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

6.1.13　出現回数

すべてのシーケンスは、count というメソッドを持っています。これは、自身の中での引数の出現回数（要素として引数を何個含んでいるかという個数）を戻り値として返す、という動作をするメソッドです。
文字列が持っている count は、引数として文字列を受け取って、引数と一致する部分シーケンスの出現回数を戻り値として返します。

```
>>> 'abc785def785ghi785jkl'.count('785')
3
>>> range(30, 40).count(36)
1
```

6.2　タプル

6.2.1　タプルの基礎

任意のオブジェクトを並べることによって作られる、順序を持つ変更不可能オブジェクトは、「タプル」(tuple) と呼ばれます。
タプルは、tuple という組み込み型から生成されるオブジェクトです。
タプルは、「丸括弧形式」(parenthesized form) と呼ばれる式を評価するか、または tuple を明示的に呼び出すことによって生成することができます。

6.2.2　丸括弧形式の書き方

丸括弧形式は、式をコンマで区切って並べて、その全体を丸括弧で囲むことによって作られます。つまり、

(式, 式, 式, ···)

という形のものを書けば、それが丸括弧形式になるということです（丸括弧は、場合によっては省略することができます）。

丸括弧形式は式ですから、評価することができます。丸括弧形式を評価すると、その中の式が評価されて、それらの式の値から構成されるタプルが生成されて、そのタプルが丸括弧形式全体の値になります。

```
>>> (583, 'namako', print, True, range(7, 20))
(583, 'namako', <built-in function print>, True, range(7, 20))
```

1 個の要素だけから構成されるタプルを生成したいときは、その要素を求める式の右側にコンマを書く必要があります。コンマを書かなかったとすると、1 個の式を丸括弧で囲んだ式ができるわけですが、それは丸括弧形式ではありません。ですから、それを評価してもタプルは生成されず、ただ単に丸括弧の中の式の値が得られるだけです。

```
>>> (583,)
(583,)
>>> (583)
583
```

0 個の要素から構成されるタプルは、「空タプル」(empty tuple) と呼ばれます。

空タプルは、0 個の式を丸括弧で囲んだ丸括弧形式を書くことによって生成することができます。

```
>>> ()
()
```

6.2.3 tuple

tuple という組み込み型は、引数としてコレクションを受け取って、そのコレクションの要素から構成されるタプルを生成して、そのタプルを戻り値として返します。

```
>>> tuple('namekuji')
('n', 'a', 'm', 'e', 'k', 'u', 'j', 'i')
>>> tuple(range(10, 60, 5))
(10, 15, 20, 25, 30, 35, 40, 45, 50, 55)
```

6.2.4 シーケンスに共通する処理：タプルの場合

タプルはシーケンスの一種ですので、第 6.1 節で説明した、すべてのシーケンスに共通する処理は、タプルに対しても実行することができます。

```
>>> a = (5, 7, 3, 7, 8, 7, 6)
>>> len(a)
7
>>> 6 in a
True
>>> 4 in a
False
>>> max(a)
8
>>> min(a)
3
>>> a[4]
8
>>> a[2:5]
(3, 7, 8)
>>> a + (2, 3, 7, 6)
(5, 7, 3, 7, 8, 7, 6, 2, 3, 7, 6)
>>> a * 2
(5, 7, 3, 7, 8, 7, 6, 5, 7, 3, 7, 8, 7, 6)
>>> a.index(8)
4
>>> a.count(7)
3
```

6.2.5 複数のオブジェクトを返す関数

Python では、関数が戻り値として返すことのできるオブジェクトの個数は、1 個だけです。しかし、複数のオブジェクトから構成される 1 個のオブジェクトを戻り値として返すことは可能です。複数のオブジェクトを戻り値として返したい場合、たとえば、それらから構成される 1 個のタプルを生成することによって、それらを戻り値として返すことができます。

次のプログラムの中で定義されている divisor という関数は、プラスの整数を受け取って、そのすべての約数から構成されるタプルを返します。

プログラムの例 divisor2.py

```
def divisor(n):
    d = ()
    for i in range(1, n+1):
        if n%i == 0:
            d += (i,)
    return d
```

実行例

```
>>> from divisor2 import *
>>> divisor(96)
(1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 96)
```

次のプログラムの中で定義されている number_to_words という関数は、プラスの整数を受け取って、それを英単語から構成されるタプルに変換した結果を返します。

プログラムの例 ntow.py

```
def digit_to_word(d):
    return ('zero', 'one', 'two', 'three', 'four', 'five',
        'six', 'seven', 'eight', 'nine')[d]

def number_to_words(n):
    w = ()
    while n > 0:
        w = (digit_to_word(n%10),) + w
        n //= 10
    return w
```

実行例

```
>>> from ntow import *
>>> number_to_words(870316)
('eight', 'seven', 'zero', 'three', 'one', 'six')
```

6.2.6 タプルに対する繰り返し

第 6.1.1 項で説明したように、すべてのコレクションは繰り返し可能オブジェクトです。したがって、タプルも、for 文を使うことによって、それを構成しているそれぞれのオブジェクトについて何らかの動作を繰り返すことができます。

次のプログラムの中で定義されている sum_of_ivisor という関数は、プラスの整数を受け取って、そのすべての約数の和を返します。

プログラムの例 sumdiv.py

```
def divisor(n):
    d = ()
    for i in range(1, n+1):
        if n%i == 0:
            d += (i,)
    return d

def sum_of_divisor(n):
    s = 0
    for i in divisor(n):
        s += i
    return s
```

実行例

```
>>> from sumdiv import *
>>> sum_of_divisor(96)
252
```

6.3 リスト

6.3.1 リストの基礎

任意のオブジェクトを並べることによって作られる、順序を持つ変更可能オブジェクトは、「リスト」(list) と呼ばれます。

任意のオブジェクトを並べることによって作られるシーケンスとしては、タプルとリストという 2 種類のものがあります。それらの相違点は、変更不可能オブジェクトかどうかということです。タプルは変更不可能オブジェクトで、リストは変更可能オブジェクトです。

リストは、`list` という組み込み型から生成されるオブジェクトです。

リストは、次の方法のうちのいずれかを使うことによって生成することができます。

- リスト表示 (list display) を使う方法。
- `list` を明示的に呼び出す方法。
- リスト内包表記 (list comprehension) を使う方法。

6.3.2 リスト表示

式をコンマで区切って並べて、その全体を角括弧で囲むことによって作られる式、つまり、

```
[式, 式, 式, ...]
```

という形の式は、「リスト表示」(list display) と呼ばれます。

リスト表示を評価すると、その中の式が評価されて、それらの式の値から構成されるリストが生成されて、そのリストがリスト表示全体の値になります。

```
>>> [583, 'namako', print, True, range(7, 20)]
[583, 'namako', <built-in function print>, True, range(7, 20)]
```

タプルとは違って、1 個の要素だけから構成されるリストを生成したいとき、その要素を求める式の右側にコンマを書く必要はありません。

```
>>> [583]
[583]
```

0 個の要素から構成されるリストは、「空リスト」(empty list) と呼ばれます。

空リストは、0 個の式を角括弧で囲んだリスト表示を書くことによって生成することができます。

```
>>> []
[]
```

6.3.3 `list`

`list` という組み込み型は、引数としてコレクションを受け取って、そのコレクションの要素から構成されるリストを生成して、そのリストを戻り値として返します。

```
>>> list('namekuji')
['n', 'a', 'm', 'e', 'k', 'u', 'j', 'i']
>>> list(range(10, 60, 5))
[10, 15, 20, 25, 30, 35, 40, 45, 50, 55]
>>> list((True, True, False, True, False))
[True, True, False, True, False]
```

6.3.4 リスト内包表記

「リスト内包表記」(list comprehension) というのは、式の一種です。それを評価すると、すでに存在するコレクションからリストが生成されて、そのリストが値として得られます。

`list` も、すでに存在するコレクションからリストを生成しますが、`list` にできることは、コレクションからリストを単純に生成することだけです。それに対して、リスト内包表記は、すで

に存在するコレクションに対して何らかの処理を加えたものからリストを生成することができます。

リスト内包表記は、基本的には、

[式$_1$ for 識別子 in 式$_2$]

と書きます。式$_2$は、コレクションを求める式でないといけません。

リスト内包表記は、次のような手順で評価されます。

(1) 式$_2$を評価する。
(2) 式$_2$の値として得られたコレクションを構成しているそれぞれの要素について、forとinのあいだに書かれた識別子をそれに束縛して、式$_1$を評価する、ということを繰り返す。
(3) 式$_1$を評価することによって得られた値から構成されるリストを生成して、それをリスト内包表記全体の値にする。

forとinのあいだに書いた識別子を、そのまま式$_1$として書くと、listを使った場合と同じように、コレクションからリストが単純に生成されます。

```
>>> [x for x in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

要素に対する処理を式$_1$として書くことによって、その処理の結果から構成されるリストを生成することができます。

```
>>> [x%3 for x in range(10)]
[0, 1, 2, 0, 1, 2, 0, 1, 2, 0]
```

リスト内包表記は、

[式$_1$ for 識別子 in 式$_2$ if 式$_3$]

と書くこともできます。式$_3$としては、何らかの条件をあらわす式を書きます。

この形のリスト内包表記は、次のような手順で評価されます。

(1) 式$_2$を評価する。
(2) 式$_2$の値として得られたコレクションを構成しているそれぞれの要素について、forとinのあいだに書かれた識別子をそれに束縛して、式$_3$を評価する、ということを繰り返す。
(3) 式$_3$の値が真だった場合は、式$_1$も評価する。
(4) 式$_1$を評価することによって得られた値から構成されるリストを生成して、それをリスト内包表記全体の値にする。

```
>>> [x for x in range(10) if x%2 == 0]
[0, 2, 4, 6, 8]
```

次のプログラムの中で定義されているdivisorという関数は、プラスの整数を受け取って、そのすべての約数から構成されるリストを返します。

プログラムの例　divisor3.py

```
def divisor(n):
    return [x for x in range(1, n+1) if n%x == 0]
```

実行例

```
>>> from divisor3 import *
>>> divisor(96)
[1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 96]
```

6.3.5　シーケンスに共通する処理：リストの場合

リストはシーケンスの一種ですので、第 6.1 節で説明した、すべてのシーケンスに共通する処理は、リストに対しても実行することができます。

```
>>> a = [5, 7, 3, 7, 8, 7, 6]
>>> len(a)
7
>>> 6 in a
True
```

```
>>> 4 in a
False
>>> max(a)
8
>>> min(a)
3
>>> a[4]
8
>>> a[2:5]
[3, 7, 8]
>>> a + [2, 3, 7, 6]
[5, 7, 3, 7, 8, 7, 6, 2, 3, 7, 6]
>>> a * 2
[5, 7, 3, 7, 8, 7, 6, 5, 7, 3, 7, 8, 7, 6]
>>> a.index(8)
4
>>> a.count(7)
3
```

6.3.6 リストの要素の置換

リストは変更可能オブジェクトですので、生成されたのちも変更を加えることができます。

リストの要素は、別のオブジェクトに置換することができます。リストの要素を置換したいときは、代入文を使います。

代入文のイコールの左側に添字表記を書くと、その添字表記で指定された要素が、イコールの右側に書かれた式の値に置換されます。

```
>>> a = [5, 7, 3, 7, 8, 7, 6]
>>> a[4] = True
>>> a
[5, 7, 3, 7, True, 7, 6]
```

代入文のイコールの左側にスライス表記を書くと、それによって指定された部分シーケンスが、イコールの右側に書かれた式の値に置換されます。この場合、イコールの右側に書く式は、コレクションを求めるものでないといけません。

```
>>> a = [5, 7, 3, 7, 8, 7, 6]
>>> a[2:5] = (True, True, False, True)
>>> a
[5, 7, True, True, False, True, 7, 6]
>>> a[2:5] = [-3, -2, -5, -4]
>>> a
[5, 7, -3, -2, -5, -4, True, 7, 6]
>>> a[2:5] = 'asari'
>>> a
[5, 7, 'a', 's', 'a', 'r', 'i', -4, True, 7, 6]
```

コロンの左右に、同じインデックスを求める式を書くことによって、そのインデックスで指定された要素の直前にコレクションを挿入する、ということもできます。

```
>>> a = [5, 7, 3, 7, 8, 7, 6]
>>> a[3:3] = 'hitode'
>>> a
[5, 7, 3, 'h', 'i', 't', 'o', 'd', 'e', 7, 8, 7, 6]
```

6.3.7 リストの要素の削除

リストの要素は、削除することができます。リストの要素を削除したいときは、「del 文」(del statement) と呼ばれる文を使います。

del 文は、

```
del 削除対象
```

と書きます。「削除対象」のところには、添字表記またはスライス表記を書きます。添字表記を書いた場合は、それによって指定された 1 個の要素が削除されます。スライス表記を書いた場合は、それによって指定された部分シーケンスが削除されます。

```
>>> a = list(range(12))
```

```
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> del a[10]
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11]
>>> del a[3:8]
>>> a
[0, 1, 2, 8, 9, 11]
```

コロンの両側の式を省略したスライス表記を書くことによって、リストを構成しているすべての要素を省略する、ということもできます。

```
>>> a = list(range(12))
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> del a[:]
>>> a
[]
```

リストが持っている remove というメソッドは、引数として受け取ったオブジェクトを、自身の中で、先頭から末尾へ向かって検索して、最初に発見した要素をリストから削除します。発見することができなかった場合は、エラーになります。

```
>>> a = [5, 7, 3, 7, 8, 7, 6]
>>> a.remove(7)
>>> a
[5, 3, 7, 8, 7, 6]
```

6.3.8　エラトステネスのふるい

2 から n までの範囲にあるすべての素数を求めるための手順としては、「エラトステネスのふるい」(sieve of Eratosthenes) と呼ばれるものがよく知られています。

エラトステネスのふるいは、次のような手順です。

(1) 2 から n までのすべての整数を並べた列を作る。

(2) 2 を i とする。

(3) 次の (4) と (5) を、i^2 が n 以下であるあいだ繰り返す。

(4) i が列の中にあるならば、その倍数のうちで列の中にあるものをすべて取り除く。ただし、i 自身は取り除かない。

(5) i に 1 を加算した整数を i とする。

この手順が終了すると、素数だけが列の中に残ります。

次のプログラムの中で定義されている eratosthenes という関数は、プラスの整数 n を受け取って、エラトステネスのふるいを使って 2 から n までの範囲にあるすべての素数を求めて、それらの素数から構成されるリストを返します。

プログラムの例　eratos.py

```
def eratosthenes(n):
    sieve = list(range(2, n+1))
    i = 2
    while i*i <= n:
        if i in sieve:
            j = i+i
            while j <= n:
                if j in sieve:
                    sieve.remove(j)
                j += i
        i += 1
    return sieve
```

実行例

```
>>> from eratos import *
>>> eratosthenes(50)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

6.4 文字列

6.4.1 この節について

この節では、文字列が持っているメソッドのうちで、重要と思われるいくつかのものを紹介したいと思います。

文字列を生成するリテラルについては第 2.1 節、文字列を数値に変換する方法や数値を文字列に変換する方法については第 2.3 節を参照してください。

6.4.2 部分文字列の探索

文字列の部分シーケンス、つまり文字列の一部分となっている文字列は、「部分文字列」(substring) と呼ばれます。

部分文字列の探索は、第 6.1.12 項で説明したように、文字列が持っている index というメソッドを使うことによって実行することができるわけですが、 find という、文字列が独自に持っているメソッドを使って実行することもできます。

index と find の相違点は、部分文字列を発見することができなかった場合の動作です。その場合、 index がエラーを出すのに対して、 find はエラーを出さないで、戻り値として −1 を返します。

```
>>> 'abc785def785ghi'.find('785')
3
>>> 'abcdefghi'.find('785')
-1
```

index と同じように、 find も、2 個目と 3 個目の引数として整数を渡すことによって、探索の範囲を限定することができます。

```
>>> 'abc785def785ghi'.find('785', 4)
9
>>> 'abc785def785ghi'.find('785', 4, 11)
-1
```

6.4.3 部分文字列の置換

文字列が持っている replace というメソッドは、引数として 2 個の文字列を受け取って、まず自身のコピーを作って、その中に部分文字列として含まれている 1 個目の引数を 2 個目の引数に置換して、そのコピーを戻り値として返します。自身は変化しません。

```
>>> 'abc785def785ghi'.replace('785', '012')
'abc012def012ghi'
```

6.4.4 文字列の分割

文字列が持っている split というメソッドは、自身を単語（連続した空白で区切られた部分文字列）に分割して、それぞれの単語から構成されるリストを戻り値として返します。

```
>>> 'the moon is a harsh mistress'.split()
['the', 'moon', 'is', 'a', 'harsh', 'mistress']
```

引数として区切り文字列を渡すと、 split は、その区切り文字列で文字列を分割します。

```
>>> '1958/09/28'.split('/')
['1958', '09', '28']
```

6.4.5 文字列の結合

文字列が持っている join というメソッドは、文字列を要素とする繰り返し可能オブジェクトを引数として受け取って、自身を区切り文字列として、引数を構成しているすべての文字列を結合することによってできる文字列を戻り値として返します。

```
>>> '/'.join(['1958', '09', '28'])
'1958/09/28'
```

第 7 章 順序なしコレクション

7.1　順序なしコレクションの基礎

7.1.1　シーケンスと順序なしコレクション

第 6.1.7 項で説明したように、コレクションは、順序を持っているものと、順序を持っていないものに分類することができて、順序を持っているコレクションは、「シーケンス」(sequence) と呼ばれます。このチュートリアルにこれまでに登場したコレクション、すなわち、範囲、文字列、タプル、リストは、いずれもシーケンスです。

順序を持っていないコレクションは、「順序なしコレクション」(unordered collection) と呼ばれます。

7.1.2　順序なしコレクションの特徴

順序なしコレクションには、次のような特徴があります。

- 要素をインデックスで識別することができない。したがって、添字表記やスライス表記を書くこともできないし、要素を探索して、そのインデックスを求めることもできない。
- 演算子の + や * を使って連結することができない。

7.1.3　順序なしコレクションの分類

順序なしコレクションは、次の 3 種類に分類することができます。

- 集合 (set)
- 不変集合 (frozen set)
- 辞書 (dictionary)

集合と不変集合については第 7.2 節で、辞書については第 7.3 節で説明することにしたいと思います。

7.2　集合と不変集合

7.2.1　集合と不変集合の基礎

集合 (set) と不変集合 (frozen set) は、どちらも、オブジェクトを単純に集めることによって作られる順序なしコレクションです。

集合と不変集合との相違点は、変更可能オブジェクトかどうかということです。集合は変更可能オブジェクトで、不変集合は変更不可能オブジェクトです。

集合も不変集合も、順序なしコレクションですので、第 7.1.2 項で指摘した特徴を持っています。また、要素の重複を許さないという特徴も持っています。つまり、1 個の集合または不変集合の中に、同一の要素が 2 個以上存在することはできない、ということです。

集合は `set` という組み込み型から生成されるオブジェクトで、不変集合は `frozenset` という組み込み型から生成されるオブジェクトです。

集合は、「集合表示」(set display) と呼ばれる式を評価するか、または、 `set` を明示的に呼び出すことによって生成することができます。

不変集合は、 `frozenset` を明示的に呼び出すことによって生成することができます。

7.2.2　集合表示の書き方

集合表示は、式をコンマで区切って並べて、その全体を中括弧で囲むことによって作られます。つまり、

```
{式, 式, 式, ···}
```

という形のものを書けば、それが集合表示になるということです。

集合表示を評価すると、その中の式が評価されて、それらの式の値から構成される集合が生成されて、その集合が集合表示全体の値になります。

```
>>> {583, 'namako', print, True, range(7, 20)}
{True, 'namako', range(7, 20), <built-in function print>, 583}
```

7.2.3　`set`

set という組み込み型は、引数としてコレクションを受け取って、そのコレクションの要素から構成される集合を生成して、その集合を戻り値として返します。

引数として渡したコレクションの中に、同一の要素が 2 個以上存在していた場合、それらの要素のうちで集合の要素になるのは 1 個だけです。

```
>>> set((5, 7, 3, 7, 8, 7, 6))
{8, 3, 5, 6, 7}
>>> set('umiushi')
{'u', 's', 'h', 'i', 'm'}
```

0 個の要素から構成される集合は、「空集合」(empty set) と呼ばれます。

空集合は、引数を何も渡さないで set を呼び出すことによって生成することができます。

```
>>> set()
set()
```

{} という式は、空集合ではなく、0 個の要素から構成される辞書を生成します。

7.2.4 frozenset

frozenset という組み込み型は、引数としてコレクションを受け取って、そのコレクションの要素から構成される不変集合を生成して、その不変集合を戻り値として返します。

```
>>> frozenset((5, 7, 3, 7, 8, 7, 6))
frozenset({8, 3, 5, 6, 7})
>>> frozenset('umiushi')
frozenset({'s', 'i', 'm', 'h', 'u'})
```

0 個の要素から構成される不変集合は、「空不変集合」(empty frozen set) と呼ばれます。

空不変集合は、引数を何も渡さないで frozenset を呼び出すことによって生成することができます。

```
>>> frozenset()
frozenset()
```

7.2.5 コレクションに共通する処理：集合の場合

集合はコレクションの一種ですので、第 6.1 節で説明した、すべてのコレクションに共通する処理は、集合に対しても実行することができます。

```
>>> a = {5, 7, 3, 8, 6, 9, 2}
>>> len(a)
7
>>> 6 in a
True
>>> 4 in a
False
>>> max(a)
9
>>> min(a)
2
```

7.2.6 集合演算子

処理の対象が集合で、処理の結果も集合であるような動作をあらわしている演算子は、「集合演算子」(set operator) と呼ばれます。

Python には、次の四つの集合演算子があります。

`a & b` a と b の共通部分 (intersection) を求める。a と b の共通部分というのは、a と b の両方に含まれる要素から構成される集合のこと。

`a | b` a と b の和集合 (union) を求める。a と b の和集合というのは、a の要素と b の要素から構成される集合のこと。

`a - b` a から b を引いた差集合 (difference) を求める。a から b を引いた差集合というのは、a には含まれるが b には含まれない要素から構成される集合のこと。

`a ^ b` a と b の対称差集合 (symmetric difference) を求める。a と b の対称差集合というのは、a と b のうちのどちらか一方のみに含まれる要素から構成される集合のこと。

```
>>> a = {1, 2, 3, 4, 5, 6}
>>> b = {4, 5, 6, 7, 8, 9}
>>> a & b
{4, 5, 6}
>>> a | b
{1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> a - b
{1, 2, 3}
>>> b - a
{8, 9, 7}
>>> a ^ b
{1, 2, 3, 7, 8, 9}
```

7.2.7　集合比較演算子

二つの集合のあいだに何らかの関係があるという条件が成り立っているかどうかを調べる比較演算子は、「集合比較演算子」(set comparison operator) と呼ばれます。

Python には、次の四つの集合比較演算子があります。

`a <= b`　a は b の部分集合 (subset) である。a は b の部分集合であるというのは、a のすべての要素が b の要素でもあるということ。

`a < b`　a は b の真部分集合 (proper subset) である。a は b の真部分集合であるというのは、a のすべての要素が b の要素でもあって、かつ、b は、a の要素ではない要素を少なくとも 1 個は含んでいるということ。

`a >= b`　a は b の上位集合 (superset) である。a は b の上位集合であるというのは、b のすべての要素が a の要素でもあるということ。

`a > b`　a は b の真上位集合 (proper superset) である。a は b の真上位集合であるというのは、b のすべての要素が a の要素でもあって、かつ、a は、b の要素ではない要素を少なくとも 1 個は含んでいるということ。

```
>>> a = {3, 4, 5}
>>> b = {2, 3, 4, 5, 6}
>>> a <= b
True
>>> b <= a
False
>>> a >= b
False
>>> b >= a
True
>>> a <= a
True
>>> a >= a
True
>>> a < a
False
>>> a > a
False
```

7.2.8　集合に対する繰り返し

第 6.1.1 項で説明したように、すべてのコレクションは繰り返し可能オブジェクトです。順序なしコレクションも例外ではありません。したがって、集合も、`for` 文を使うことによって、それを構成しているそれぞれのオブジェクトについて何らかの動作を繰り返すことができます。ただし、識別子がそれぞれの要素に束縛される順序は不定です。

```
>>> a = {5, 7, 3, 8, 6, 9, 2}
>>> for x in a:
...     print(x, end=' ')
...
2 3 5 6 7 8 9 >>>
```

7.2.9 ハッシュ可能性

タプルとリストは、任意のオブジェクトを要素として持つことができます。それに対して、集合には、要素として持つことができないオブジェクトが存在します。

オブジェクトが集合の要素になることができるためには、そのオブジェクトが「ハッシュ可能性」(hashability) という性質を持っている必要があります。

ハッシュ可能性というのは、「ハッシュ値」(hash value) とよばれる整数を持っていて、かつ、他のオブジェクトとのあいだで、等しいかどうかの比較ができる、という性質のことです。

ハッシュ可能性を持つオブジェクトは「ハッシュ可能オブジェクト」(hashable object) と呼ばれ、ハッシュ可能性を持たないオブジェクトは「ハッシュ不可能オブジェクト」(unhashable object) と呼ばれます。

基本的には、スカラー、文字列、範囲などの変更不可能オブジェクトはハッシュ可能オブジェクトで、リストなどの変更可能オブジェクトはハッシュ不可能オブジェクトです。

リストはハッシュ不可能オブジェクトですので、要素としてリストを含む集合を作るということはできません。無理に作ろうとすると、エラーになります。

```
>>> {5, 7, [3, 8, 2], 4, 6}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

タプルは、そのすべての要素が変更不可能オブジェクトならばハッシュ可能オブジェクトですが、要素として変更可能オブジェクトを 1 個でも持っている場合はハッシュ不可能オブジェクトになります。

```
>>> {5, 7, (3, 8, 2), 4, 6}
{(3, 8, 2), 4, 5, 6, 7}
>>> {5, 7, (3, [8, 2]), 4, 6}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

集合と辞書も、ハッシュ不可能オブジェクトです。したがって、それらを要素として含む集合を作ることはできません。それに対して、不変集合はハッシュ可能オブジェクトですので、要素として不変集合を含む集合を作ることは可能です。

```
>>> {5, 7, {3, 8, 2}, 4, 6}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
>>> {5, 7, frozenset((3, 8, 2)), 4, 6}
{frozenset({8, 2, 3}), 4, 5, 6, 7}
```

7.3 辞書

7.3.1 辞書の基礎

「辞書」(dictionary) と呼ばれるオブジェクトは、集合と同じように、オブジェクトを集めることによって作られる順序なしコレクションです。

集合と辞書との相違点は、要素の構造です。集合の個々の要素は単独のオブジェクトですが、辞書の要素は、2 個のオブジェクトがペアになったものです。

辞書の要素を構成しているペアのオブジェクトのそれぞれは、「キー」(key) と「値」(datum) と呼ばれます。

キーは、個々の要素を識別するために使われるオブジェクトです。したがって、1 個の辞書の中に、同じキーを持つ要素が 2 個以上存在することはできません。それに対して、値は、要素の識別には使われませんので、1 個の辞書の中に、同じ値を持つ要素が 2 個以上存在することも可能です。

辞書は、現実の世界に存在する辞書に、よく似ています。現実の辞書は、見出し語と、その見出し語についての説明とをペアにした項目から構成されています。キーというのは見出し語に相当して、値というのは見出し語についての説明に相当します。そして、それぞれの項目は、見出し語によって識別されます。

辞書の要素の値にすることができるのは、任意のオブジェクトです。それに対して、辞書の要素のキーにすることができるのは、ハッシュ可能オブジェクトだけです。ハッシュ不可能オブジェクトを辞書の要素のキーにすることはできません。

辞書は、dict という組み込み型から生成されるオブジェクトです。

辞書は、「辞書表示」(dictionary display) と呼ばれる式を評価することによって生成することができます。

7.3.2　辞書表示の書き方

辞書はキーと値のペアから構成されますので、辞書表示は、キーと値のペアをあらわす記述から構成されることになります。キーと値のペアをあらわす記述は、「キー値ペア」(key/datum pair) と呼ばれます。

キー値ペアは、

```
式: 式
```

というように、コロンの左右に式を書いたものです。コロンの左にはキーを求める式、右側には値を求める式を書きます。たとえば、

```
'namako': 707
```

というキー値ペアを書くことによって、namako という文字列がキーで、707 という整数が値であるような辞書の要素を記述することができます。

コロンとその右側の式とのあいだには、通常、1 個の空白を書きます。ただしこれは、文法でそのように決められているわけではなくて、あくまでスタイルの問題です。

辞書表示は、キー値ペアをコンマで区切って並べて、その全体を中括弧で囲むことによって作られます。つまり、

```
{キー値ペア, キー値ペア, キー値ペア, ···}
```

という形のものを書けば、それが辞書表示になるということです。

辞書表示を評価すると、その中の式が評価されて、それらの式の値から構成される辞書が生成されて、その辞書が辞書表示全体の値になります。

```
>>> {'namako': 707, 532: {8, 4, 6}, (1, 9): [3, 0, 8]}
{532: {8, 4, 6}, (1, 9): [3, 0, 8], 'namako': 707}
```

0 個の要素から構成される辞書は、「空辞書」(empty dictionary) と呼ばれます。

空辞書は、0 個のキー値ペアを中括弧で囲んだ辞書表示を書くことによって生成することができます。

```
>>> {}
{}
```

7.3.3　コレクションに共通する処理：辞書の場合

辞書はコレクションの一種ですので、第 6.1 節で説明した、すべてのコレクションに共通する処理は、辞書に対しても実行することができます。ただし、in と not in が調べるのは、キーが辞書に帰属するかどうかです。また、max と min のそれぞれが求めるのは、キーの最大値と最小値です。

```
>>> a = {'a': 8, 'w': 2, 'g': 7, 'd': 6, 'm': 4}
>>> len(a)
5
>>> 'd' in a
True
>>> 'e' in a
False
>>> max(a)
'w'
>>> min(a)
'a'
```

7.3.4　辞書の添字表記

辞書が持っている特定の要素は、添字表記を書くことによって指定することができます。

辞書の添字表記は、シーケンスの添字表記と同じように、

$a[k]$

と書きます。a は辞書を求める式で、k は要素のキーを求める式です。

辞書の添字表記も、シーケンスの添字表記と同じように、式として評価することができます。辞書の添字表記を式として評価すると、その値として、指定されたキーを持つ要素の値が得られます。指定されたキーを持つ要素が存在しない場合は、エラーになります。

```
>>> a = {'w': 2, 'g': 7, 'd': 6}
>>> a['d']
6
>>> a['k']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'k'
```

キーで指定された要素の値は、辞書が持っている get というメソッドを使うことによって求めることもできます。get は、オブジェクトを引数として受け取って、それをキーとして持つ要素の値を戻り値として返します。

```
>>> a = {'w': 2, 'g': 7, 'd': 6}
>>> a.get('d')
6
```

指定されたキーを持つ要素が存在しない場合、添字表記の評価では、エラーになりますが、get はエラーにならず、戻り値として None を返します。get に 2 個目の引数を渡すと、指定されたキーを持つ要素が存在しない場合、None の代わりに 2 個目の引数を戻り値として返します。

```
>>> a = {'w': 2, 'g': 7, 'd': 6}
>>> a.get('k', 'not found')
'not found'
```

7.3.5 辞書の要素の置換

辞書は変更可能オブジェクトですので、生成されたのちも変更を加えることができます。

辞書の要素の値は、別のオブジェクトに置換することができます。辞書の要素の値を置換したいときは、代入文を使います。

代入文のイコールの左側に添字表記を書くと、その添字表記で指定された要素の値が、イコールの右側に書かれた式の値に置換されます。

```
>>> a = {'w': 2, 'g': 7, 'd': 6}
>>> a['d'] = True
>>> a
{'g': 7, 'w': 2, 'd': True}
```

辞書の要素を置換する代入文を実行したときに、角括弧の中の式の値をキーとして持つ要素が存在しなかった場合は、その式の値をキーとして、イコールの右側に書かれた式の値を値とする要素が辞書に追加されます。

```
>>> a = {'w': 2, 'g': 7, 'd': 6}
>>> a['m'] = False
>>> a
{'m': False, 'g': 7, 'w': 2, 'd': 6}
```

7.3.6 要素の削除

辞書の要素は、del 文を使うことによって、削除することができます。

辞書の要素を削除する del 文は、リストの要素を削除する場合と同じように、

del 削除対象

と書きます。「削除対象」のところには、削除したい要素を指定する添字表記を書きます。

```
>>> a = {'w': 2, 'g': 7, 'd': 6}
>>> del a['g']
>>> a
{'w': 2, 'd': 6}
```

辞書が持っている clear というメソッドは、辞書を構成しているすべての要素を削除します。

```
>>> a = {'w': 2, 'g': 7, 'd': 6}
```

```
>>> a.clear()
>>> a
{}
```

7.3.7　辞書ビューオブジェクト

辞書は、keys、values、items というメソッドを持っています。これらのメソッドは、「辞書ビューオブジェクト」(dictionary view object) と呼ばれるオブジェクトを戻り値として返します（引数は受け取りません)。辞書ビューオブジェクトというのは、自身を生成した辞書に関連づけられていて、辞書が変更されると、それに連動して自身も変更されるオブジェクトのことです。

keys、values、items のそれぞれが返すのは、次のような辞書ビューオブジェクトです。

keys	辞書のすべてのキーから構成される辞書ビューオブジェクト。
values	辞書のすべての値から構成される辞書ビューオブジェクト。
items	辞書のすべての要素から構成される辞書ビューオブジェクト。

辞書ビューオブジェクトは、tuple を呼び出すことによってタプルに、list を呼び出すことによってリストに変換することができます。

```
>>> a = {'w': 2, 'g': 7, 'd': 6}
>>> tuple(a.keys())
('g', 'd', 'w')
>>> list(a.values())
[7, 6, 2]
>>> tuple(a.items())
(('g', 7), ('d', 6), ('w', 2))
>>> list(a.items())
[('g', 7), ('d', 6), ('w', 2)]
```

辞書ビューオブジェクトは、それに関連づけられている辞書を変更すると、それに連動して変更されます。

```
>>> a = {'w': 2, 'g': 7, 'd': 6}
>>> b = a.items()
>>> tuple(b)
(('g', 7), ('d', 6), ('w', 2))
>>> a['d'] = True
>>> tuple(b)
(('g', 7), ('d', True), ('w', 2))
```

7.3.8　度数分布

何かの度数分布を求めたいときは、辞書を使うと便利です。

次のプログラムの中で定義されている frequency という関数は、文字列を受け取って、その文字列を構成している文字の度数分布を示す辞書を返します。

プログラムの例　frequency.py

```
def frequency(s):
    fd = {}
    for c in s:
        if c in fd:
            fd[c] += 1
        else:
            fd[c] = 1
    return fd
```

実行例

```
>>> from frequency import *
>>> frequency('sakatanokintoki')
{'t': 2, 'a': 3, 's': 1, 'n': 2, 'o': 2, 'i': 2, 'k': 3}
```

第 8 章　クラス

8.1 クラスの基礎

8.1.1 クラスについての復習

この章では、クラスについて説明するわけですが、まず最初に、クラスについてこれまでに説明したことを復習しておくことにしましょう。

第 2.3.2 項で説明したように、オブジェクトを生成するという動作をする呼び出し可能オブジェクトは、「クラス」(class) と呼ばれます。クラスは、何らかのオブジェクトを生成して、そのオブジェクトを戻り値として返します。Python のオブジェクトは、例外なく、何らかのクラスから生成されます。

第 2.3.10 項で説明したように、Python では、「クラス定義」(class definition) と呼ばれる文をプログラムの中に書くことによって、クラスを自由に定義することができます。プログラムの中にクラス定義を書くことによって定義されたクラスは、「ユーザー定義クラス」(user-defined class) と呼ばれます。

「ユーザー定義クラス」の対義語は、「組み込み型」です。「組み込み型」(built-in type) というのは、Python の処理系に組み込まれているクラスのことです。

8.1.2 クラスとオブジェクトとの関係

クラスとオブジェクトとの関係は、鋳型と鋳物との関係に似ています。

「鋳物」というのは、高熱で溶けた金属を冷やして固めることによって製造される金属製品のことです。鋳物を製造するとき、溶けた金属は、「鋳型」と呼ばれるものの中にある空洞に流し込まれます。そうすることによって、鋳物は、鋳型の中の空洞と同じ形を持つことになります。そして、ひとつの鋳型からは、同じ形を持つ鋳物をいくつでも製造することができます。

鋳物が鋳型から製造されるのと同じように、オブジェクトはクラスから生成されます。ひとつの鋳型から製造されたすべての鋳物が同じ形を持つのと同じように、ひとつのクラスから生成されたすべてのオブジェクトは、同じメソッドを持つことになります。

8.1.3 組み込み型

このチュートリアルにこれまでに登場したオブジェクトは、すべて、組み込み型から生成されたものです。

すべての組み込み型には、あらかじめ識別子が束縛されています。次の表は、このチュートリアルにこれまでに登場したオブジェクトを生成する組み込み型のうちで主要なものについて、それに束縛されている識別子を示しています。

整数	`int`	範囲	`range`
浮動小数点数	`float`	タプル	`tuple`
文字列	`str`	リスト	`list`
真偽値	`bool`	集合	`set`
ユーザー定義関数	`function`	辞書	`dict`

8.1.4 クラスを生成する組み込み型

クラスというのはオブジェクトの一種ですから、ほかのオブジェクトと同じように、何らかのクラスから生成されます。

クラスを生成するクラスは組み込み型で、`type` という識別子がその組み込み型に束縛されています。

`type` は、引数としてオブジェクトを受け取って、そのオブジェクトを生成したクラスを戻り値として返します。

```
>>> type(87)
<class 'int'>
>>> type(3.14)
<class 'float'>
>>> type('namako')
<class 'str'>
```

クラスを生成したクラスが `type` だということも、`type` を呼び出すことによって確かめることができます。

```
>>> type(int)
<class 'type'>
```

type もオブジェクトですから、何らかのクラスから生成されます。type を生成するのは、自分自身です。

```
>>> type(type)
<class 'type'>
```

8.2 クラス定義

8.2.1 クラス定義の基礎

クラスを生成して、識別子をそのクラスに束縛することを、クラスを「定義する」(define) と言います。

クラスは、「クラス定義」(class definition) と呼ばれる文を書くことによって定義することができます。クラス定義は、クラスを定義するという動作を意味する文です。

8.2.2 基本的なクラス定義

クラス定義は、基本的には、

```
class 識別子:
    ブロック
```

と書きます。クラス定義を実行すると、新しいクラスが生成されて、「識別子」のところに書かれた識別子がそのクラスに束縛されます。

それでは、クラス定義をインタラクティブシェルに入力してみましょう。

```
>>> class Hitode:
...     print('hitode')
...
hitode
```

これによって、新しいクラスが生成されて、Hitode という識別子がそのクラスに束縛されます。クラス定義が実行されると、それと同時に、その中のブロックも実行されますので、この場合は、hitode という文字列が出力されています。

なお、ユーザー定義クラスに束縛する識別子は、この Hitode のように、その先頭の文字を大文字にします（これは文法で決まっていることではなくて、プログラムを読みやすくするための約束事です）。

8.2.3 オブジェクトの生成

クラスというのは呼び出し可能オブジェクトですから、呼び出しを書くことによってクラスを呼び出すことができます。クラスを呼び出すと、オブジェクトが生成されて、そのオブジェクトが戻り値として返ってきます。

先ほど定義した Hitode クラスも、呼び出すことによってオブジェクトを生成することができます。インタラクティブシェルを使って試してみましょう。

```
>>> h = Hitode()
>>> h
<__main__.Hitode object at 0x00C40830>
>>> type(h)
<class '__main__.Hitode'>
```

8.2.4 何の動作もしない文

先ほど定義した Hitode というクラスから生成されたオブジェクトは、クラス定義の中にメソッドを作る記述を書いていませんが、メソッドをまったく持っていないわけではなくて、いくつかのデフォルトのメソッドを持っています。

Hitode のような、デフォルトのメソッドだけを持つオブジェクトを生成するクラスを定義したいときは、その中のブロックとして、何の動作もしない文を書くのが普通です。

Python には、「pass 文」(pass statement) と呼ばれる、何の動作もしない文があります。pass 文は、

```
pass
```

と書くだけの文です。

それでは、ブロックとしてpass文を書いたクラス定義をインタラクティブシェルに入力してみましょう。

```
>>> class Kurage:
...     pass
...
>>> Kurage()
<__main__.Kurage object at 0x00C4A310>
```

8.3 メソッドの定義

8.3.1 メソッドの定義の基礎

第8.1.2項で説明したように、クラスというのは、オブジェクトを生成する鋳型のようなものだと考えることができます。

オブジェクトの中のメソッドは、クラスという鋳型の中にあるメソッドの鋳型から生成されます。ですから、メソッドの鋳型をクラスの中に作ることによって、メソッドを持つオブジェクトを生成するクラスを作ることができます。

メソッドの鋳型をクラスの中に作って、識別子をその鋳型に束縛することを、メソッドを「定義する」(define) と言います。

第3.3節で紹介した関数定義という文は、関数を定義する場合だけではなくて、メソッドを定義する場合にも使われます。ただし、メソッドを定義したい場合、そのための関数定義は、次の二つの条件を満足する必要があります。

- クラス定義のブロックの中に書かれていなければならない。
- 少なくとも1個の引数を受け取るように書かれていないといけない。

メソッドは、少なくとも1個の引数を受け取ります。ですから、それを定義する関数定義には、少なくとも1個の仮引数を書く必要があります。文法的には、仮引数としてどんな識別子を使ってもかまわないわけですが、メソッドの1個目の仮引数としては、selfという識別子を使うことになっています。

それでは、メソッドを定義する関数定義を持つクラス定義を、インタラクティブシェルに入力してみましょう。

```
>>> class Namako:
...     def what(self):
...         print('namako')
...
```

このクラス定義によって定義されたNamakoというクラスは、メソッドを生成するwhatという鋳型を持っています。ですから、Namakoから生成されたオブジェクトは、whatから生成されたメソッドを持つことになります。

今後は、「○○というメソッドの鋳型から生成されたメソッド」のことを、誤解のおそれのない限り、ただ単に「○○」と呼ぶことにしたいと思います。

8.3.2 メソッドの呼び出し

メソッドが受け取る1個目の引数は、メソッドに暗黙的に渡されます。ですから、先ほど定義した、Namakoクラスのオブジェクトが持っているwhatというメソッドを呼び出す場合、明示的に引数を渡す必要はありません。

第2.3.5項で説明したように、メソッドを求めたいときは、

式 . メソッド名

という形の式を書きます。この形の式を評価すると、ドットの左側の式を評価することによって得られたオブジェクトが持っている、メソッド名で指定されたメソッドが、その値として得られます。ですから、nという変数がNamakoクラスのオブジェクトに束縛されているとすると、

```
n.what
```

という式を評価することによって、n が持っている what というメソッドを求めることができます。呼び出し可能オブジェクトは、それを求める式の右側に丸括弧を書くことによって呼び出すことができるわけですから、

```
n.what()
```

という式を評価することによって、n が持っている what を呼び出すことができます。

それでは、Namako クラスからオブジェクトを生成して、それが持っている what というメソッドを呼び出してみましょう。

```
>>> n = Namako()
>>> n.what()
namako
```

8.3.3 メソッドが受け取る 1 個目の引数

すべてのメソッドは、暗黙的に 1 個目の引数を受け取ります。その引数というのは、そのメソッドを持っているオブジェクト自身です。第 8.3.1 項で、メソッドの 1 個目の仮引数としては self という識別子を使うことになっていると説明しましたが、 self という識別子を使う理由は、それがオブジェクト自身に束縛されることになるからです。

それでは、オブジェクト自身を戻り値として返すメソッドを持つオブジェクトを生成するクラスを定義してみましょう。

```
>>> class Umiushi:
...     def get_self(self):
...         return self
...
>>> u = Umiushi()
>>> u
<__main__.Umiushi object at 0x00C4FB30>
>>> u.get_self()
<__main__.Umiushi object at 0x00C4FB30>
```

8.3.4 2 個以上の引数を受け取るメソッド

メソッドを定義する関数定義が 2 個以上の仮引数を持っている場合、2 個目以降の仮引数は、明示的に渡された引数に束縛されます。

```
>>> class Hamaguri:
...     def add_hamaguri(self, s):
...         return s + 'hamaguri'
...
>>> h = Hamaguri()
>>> h.add_hamaguri('hoge')
'hogehamaguri'
```

8.3.5 属性

個々のオブジェクトは、「属性」(attribute) と呼ばれる識別子を持つことができます。属性というのは、個々のオブジェクトに専属の識別子だと考えることができます。「個々のオブジェクトに専属」というのは、識別子がオブジェクトに束縛されるという関係が、個々のオブジェクトごとに別々だということです。

たとえば、A と B という二つのオブジェクトがあって、そのどちらも size という属性を持っているとしましょう。この場合、A の size を 53 に束縛して、B の size を 78 に束縛するというように、それぞれの属性を別のオブジェクトに束縛することができます。

プログラムの中で属性を扱いたいときは、

式 . 属性

という形のものを書きます。この形のものは、式の値として得られたオブジェクトが持っている属性を指定する記述です。この形のものを式として評価すると、属性が束縛されているオブジェクトが値として得られます。

代入文のイコールの左側に、属性を指定する記述を書くことによって、属性をオブジェクトに束縛することができます。

```
>>> class Namekuji:
```

```
...     pass
...
>>> n1 = Namekuji()
>>> n2 = Namekuji()
>>> n1.size = 37
>>> n2.size = 84
>>> n1.size
37
>>> n2.size
84
```

メソッドの中でも、self を使うことによって、属性を扱うことができます。

```
>>> class Sazae:
...     def get_size(self):
...         return self.size
...
>>> s1 = Sazae()
>>> s2 = Sazae()
>>> s1.size = 48
>>> s2.size = 73
>>> s1.get_size()
48
>>> s2.get_size()
73
```

8.3.6 オブジェクトの初期化

クラスからオブジェクトが生成されるときに、そのオブジェクトが自動的に初期化されるようにしたい、ということがしばしばあります。Python では、「初期化メソッド」(initialization method) と呼ばれる特殊なメソッドを定義しておくことによって、オブジェクトの初期化を自動化することができます。

初期化メソッドというのは、__init__という識別子が束縛されているメソッドのことです。この識別子をメソッドに束縛しておくと、そのメソッドは、クラスからオブジェクトが生成されるときに自動的に呼び出されます。ですから、オブジェクトの初期化に必要な動作をするように初期化メソッドを定義しておけば、オブジェクトの初期化が自動的に実行されるようになります。

```
>>> class Fujitsubo:
...     def __init__(self):
...         self.size = 67
...
>>> f = Fujitsubo()
>>> f.size
67
```

__init__は、戻り値として None を返すように定義する必要があります。ですから、__init__の定義の中に return 文を書く必要はありません（第 3.6.2 項で説明したように、return 文を実行しないで終了した関数は None を返すわけですが、これはメソッドも同じです）。

ちなみに、None 以外の戻り値を返す__init__を定義すると、オブジェクトを生成したときにエラーになります。

```
>>> class Hoya:
...     def __init__(self):
...         return 35
...
>>> h = Hoya()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() should return None, not 'int'
```

クラスに引数を渡すと、それらの引数は__init__に渡されます。

```
>>> class Kamenote:
...     def __init__(self, size):
...         self.size = size
...
>>> k = Kamenote(48)
>>> k.size
```

```
48
```

次のプログラムの中で定義されている Saifu というクラスは、財布をシミュレートするオブジェクトを生成します。

Saifu クラスは、財布に入れるお金の金額を引数として受け取ります。財布の中のお金は、dashiire というメソッドを呼び出すことによって出し入れすることができます。dashiire に渡す引数は、出し入れするお金の金額です。プラスの金額を渡すと財布の中のお金が増えて、マイナスの金額を渡すとお金が減ります。

プログラムの例　saifu.py

```
class Saifu:
    def __init__(self, kingaku):
        self.kingaku = kingaku
    def dashiire(self, kingaku):
        self.kingaku += kingaku
```

実行例

```
>>> from saifu import *
>>> s = Saifu(3000)
>>> s.kingaku
3000
>>> s.dashiire(800)
>>> s.kingaku
3800
>>> s.dashiire(-1200)
>>> s.kingaku
2600
```

8.4　派生クラス

8.4.1　基底クラスと派生クラス

第 8.1.2 項で説明したように、クラスというのはオブジェクトを生成する鋳型のようなものです。しかし、鋳型のようなものというのは、クラスというものの側面のひとつにすぎません。クラスには、もうひとつの別の側面があります。それは、「分類体系の部品」という側面です。

さまざまなクラスの全体は、生物などの分類体系と同じように、木の形をした構造を持っています。この木は、本物の木とは上と下とが逆になっていて、いちばん上に根があって、下に向かって枝が伸びています。木を構成するそれぞれのクラスは、上にあるものほど一般的で、下にあるものほど特殊です。

Python では、すでに存在するクラスを特殊化することによって新しいクラスを作るということができます。クラスを特殊化するというのは、メソッドを追加するということです。

B というクラスが、A というクラスを特殊化することによって作られたものだとするとき、A は、B の「基底クラス」(base class) と呼ばれ、B は、A の「派生クラス」(derived class) と呼ばれます。

8.4.2　派生クラスの定義

すでに存在するクラスを特殊化した新しいクラスを定義したいとき、つまり、派生クラスを定義したいときは、

```
class 識別子 (基底クラス名):
    ブロック
```

という形のクラス定義を書きます。そうすると、「基底クラス名」のところに書いた識別子に束縛されているクラスを特殊化した派生クラスが定義されます。

8.4.3　継承

オブジェクト指向をサポートするプログラミング言語の大多数は、基底クラスが持っている変数やメソッドの鋳型が派生クラスに自動的に受け継がれるという機能を持っていて、そのような機能は「継承」(inheritance) と呼ばれます。そして、継承によって変数やメソッドの鋳型を受け継ぐことを、変数やメソッドの鋳型を「継承する」(inherit) と言います。

Python も、継承という機能を備えているプログラミング言語のひとつです。

次のプログラムの中で定義されている Ningen は、人間をあらわすオブジェクトを生成するクラスです。そして、 Rikishi は、 Ningen クラスの派生クラスで、力士をあらわすオブジェクトを生成します。

プログラムの例 rikishi.py

```
class Ningen:
    def set_namae(self, namae):
        self.namae = namae

class Rikishi(Ningen):
    def set_shikona(self, shikona):
        self.shikona = shikona
```

実行例

```
>>> from rikishi import *
>>> r = Rikishi()
>>> r.set_namae('Yamanaka')
>>> r.namae
'Yamanaka'
>>> r.set_shikona('Hogenoumi')
>>> r.shikona
'Hogenoumi'
```

このように、 Rikishi クラスのオブジェクトは、 Ningen クラスから継承した set_namae というメソッドと、 Rikishi クラスで定義した set_shikona というメソッドを持っています。

8.4.4 オーバーライド

新しいクラスを定義するとき、新しいメソッドを追加するだけではなくて、基底クラスで定義されているメソッドの動作を変更したい、ということがしばしばあります。基底クラスのメソッドの動作を派生クラスで変更したいときは、基底クラスで定義されているメソッドと同じ名前のメソッドを派生クラスで定義します。このことを、基底クラスのメソッドを「オーバーライドする」(override) と言います。

次のプログラムは、まず Sakana というクラスを定義して、次に、 Maguro というクラスを、 Sakana を基底クラスとする派生クラスとして定義しています。 Maguro クラスは、 Sakana クラスで定義されている what というメソッドの動作を、それをオーバーライドすることによって変更しています。

プログラムの例 maguro.py

```
class Sakana:
    def what(self):
        return 'sakana'

class Maguro(Sakana):
    def what(self):
        return 'maguro'
```

実行例

```
>>> from maguro import *
>>> s = Sakana()
>>> s.what()
'sakana'
>>> m = Maguro()
>>> m.what()
'maguro'
```

8.4.5 オーバーライドされたメソッドの呼び出し

オーバーライドされた基底クラスのメソッドは、派生クラスのメソッドによって隠されることになりますが、派生クラスのメソッドの中から呼び出すことは可能です。

オーバーライドされた基底クラスのメソッドを派生クラスのメソッドから呼び出したいときは、

super という組み込み関数を使います。派生クラスのメソッドを定義する関数定義の中で super を呼び出すと、 super は、基底クラスのメソッドを持つオブジェクトを戻り値として返します。ですから、 what という名前のメソッドが派生クラスの what によってオーバーライドされているとするとき、派生クラスのメソッドを定義する関数定義の中に、

```
super().what()
```

と書くと、基底クラスの what が呼び出されることになります。

次のプログラムは、まず Konchuu というクラスを定義して、次に、 Tonbo というクラスを、 Konchuu を基底クラスとする派生クラスとして定義しています。 Tonbo クラスは、 Konchuu クラスで定義されている what というメソッドの動作を、それをオーバーライドすることによって変更しています。

Tonbo クラスの what は、オーバーライドによって隠されている基底クラスの what を、 super を使うことによって呼び出しています。

プログラムの例　tonbo.py

```
class Konchuu:
    def what(self):
        return 'konchuu'

class Tonbo(Konchuu):
    def what(self):
        return super().what() + '/tonbo'
```

実行例

```
>>> from tonbo import *
>>> t = Tonbo()
>>> t.what()
'konchuu/tonbo'
```

初期化メソッドというのは__init__という識別子が束縛されているメソッドのことですから、派生クラスと基底クラスの両方で初期化メソッドを定義すると、派生クラスの初期化メソッドは基底クラスの初期化メソッドをオーバーライドすることになります。しかし、そのような場合でも、 super を使えば、派生クラスの初期化メソッドから基底クラスの初期化メソッドを呼び出すことができます。

次のプログラムは、まず Ningen というクラスを定義して、次に、それを基底クラスとする派生クラスとして Rikishi というクラスを定義しています。

Ningen クラスの初期化メソッドは Rikishi クラスの初期化メソッドによってオーバーライドされていますが、 Rikishi クラスの初期化メソッドは、 namae という属性を初期化するために、 super を使って Ningen クラスの初期化メソッドを呼び出しています。

プログラムの例　rikishi2.py

```
class Ningen:
    def __init__(self, namae):
        self.namae = namae

class Rikishi(Ningen):
    def __init__(self, namae, shikona):
        super().__init__(namae)
        self.shikona = shikona
```

実行例

```
>>> from rikishi2 import *
>>> r = Rikishi('Yamanaka', 'Hogenoumi')
>>> r.namae
'Yamanaka'
>>> r.shikona
'Hogenoumi'
```

第 9 章　ファイル

9.1 オープンとクローズ

9.1.1 オープンとクローズの基礎

この章では、ファイルに対する操作を実行するプログラムについて説明したいと思います。

ファイルからデータを読み込んだり、ファイルにデータを書き込んだりするためには、それに先立って、そのための準備をする必要があります。ファイルに対する読み書きのための準備をすることを、ファイルを「オープンする」(open) と言います。

ファイルに対して読み書きを実行したときは、それが終わったのち、その後始末をする必要があります。ファイルに対する読み書きが終わったのちにその後始末をすることを、ファイルを「クローズする」(close) と言います。

9.1.2 ファイルをオープンする組み込み関数

ファイルは、 `open` という組み込み関数を呼び出すことによってオープンすることができます。

`open` は、1 個目の引数として、オープンするファイルのパス名を受け取って、2 個目の引数として、ファイルのオープンモードを指定する文字列を受け取ります。

ファイルの「オープンモード」(open mode) というのは、どのような処理を実行するためにファイルをオープンするのかということです。オープンモードは、次のような文字を組み合わせることによってできる文字列で指定します。

`r` 読み込みのためにオープンする。

`w` 書き込みのためにオープンする。ファイルが存在しない場合は、空のファイルを作成する。すでにファイルが存在する場合は、そのファイルを空にする。

`a` 書き込みのためにオープンする。ファイルが存在しない場合は、空のファイルを作成する。すでにファイルが存在する場合は、その末尾にデータを追加する。

`t` テキストモードでオープンする（デフォルト）。

`b` バイナリーモードでオープンする。

たとえば、 `wb` は、書き込みのためにバイナリーモードでオープンするという意味になります。`t` は省略することができますので、 `w` は、書き込みのためにテキストモードでオープンするという意味になります。

2 個目の引数は、デフォルトが `rt` です。したがって、読み込みのためにテキストモードでオープンする場合、2 個目の引数を渡す必要はありません。

`open` は、戻り値として、「ファイルオブジェクト」(file object) と呼ばれるオブジェクトを返します。ファイルに対する読み込みや書き込みは、このオブジェクトが持っているメソッドを呼び出すことによって実行されます。

9.1.3 ファイルをクローズする組み込み関数

ファイルは、ファイルオブジェクトが持っている `close` というメソッドを呼び出すことによってクローズすることができます。

`close` には、引数を渡す必要はありません。

9.1.4 ストリーム位置

ファイルに対する読み書きは、「ストリーム位置」(stream position) と呼ばれる位置に対して実行されます。

`r` でファイルをオープンした場合、オープンした直後のストリーム位置は、ファイルの先頭です。

`w` でファイルをオープンした場合も、ファイルは空になっているわけですから、最初のストリーム位置はファイルの先頭です。それに対して、すでに存在するファイルを `a` でオープンした場合、最初のストリーム位置はファイルの末尾になります。

データの読み書きをするメソッドは、ストリーム位置に対して読み書きを実行して、そののち、次に読み書きを実行するべき位置へストリーム位置を移動させます。

ファイルの末尾というストリーム位置は、「ファイルの終わり」(end of file) と呼ばれます。

9.2　ファイルからの読み込み

9.2.1　ファイルからデータを読み込むメソッド

ファイルオブジェクトは、read というメソッドを持っています。このメソッドは、引数を何も渡さないで呼び出すと、ストリーム位置からファイルの終わりまでのデータをファイルから読み込んで、読み込んだデータを文字列として返します。

次のプログラムの中で定義されている read_file という関数は、パス名を受け取って、それによって指定されるテキストファイルの内容を出力します。

プログラムの例　readfile.py

```
def read_file(filename):
    f = open(filename)
    print(f.read(), end='')
    f.close()
```

テキストファイルの例　puellae.txt

```
Kaname Madoka
Akemi Homura
Miki Sayaka
Tomoe Mami
```

実行例

```
>>> from readfile import *
>>> read_file('puellae.txt')
Kaname Madoka
Akemi Homura
Miki Sayaka
Tomoe Mami
```

引数としてプラスの整数 n を渡して read を呼び出すと、read は、n バイトのデータを読み込んで、ストリーム位置を n バイト先へ移動させて、読み込んだデータを文字列として返します。

read は、ストリーム位置がファイルの終わりだった場合、空文字列を戻り値として返します。

次のプログラムの中で定義されている space_to_dot という関数は、パス名を受け取って、それによって指定されるテキストファイルの内容に含まれているすべての空白をドットに変換した結果を出力します。

プログラムの例　space2dot.py

```
def space_to_dot(filename):
    f = open(filename)
    eof = False
    while not eof:
        c = f.read(1)
        if c == '':
            eof = True
        else:
            if c != ' ':
                print(c, end='')
            else:
                print('.', end='')
    f.close()
```

実行例

```
>>> from space2dot import *
>>> space_to_dot('puellae.txt')
Kaname.Madoka
Akemi.Homura
Miki.Sayaka
Tomoe.Mami
```

9.2.2 読み込んだデータをリストにして返すメソッド

ファイルオブジェクトは、readlines というメソッドを持っています。このメソッドは、ストリーム位置からファイルの終わりまでのデータをファイルから読み込んで、読み込んだデータを、それぞれの行から構成されるリストにして返します。

次のプログラムの中で定義されている space_to_dot という関数は、パス名を受け取って、それによって指定されるテキストファイルの内容を、それぞれの行から構成されるリストの形で出力します。

プログラムの例 linecount.py

```
def linecount(filename):
    f = open(filename)
    lines = f.readlines()
    print('行数は' + str(len(lines)) + 'です。')
    f.close()
```

実行例

```
>>> from linecount import *
>>> linecount('puellae.txt')
行数は 4 です。
```

9.2.3 データを 1 行だけ読み込むメソッド

ファイルオブジェクトは、readline というメソッドを持っています。このメソッドは、ストリーム位置から行末までのデータをファイルから読み込んで、ストリーム位置を次の行の先頭へ移動させて、読み込んだデータ（改行文字を含む）を文字列として返します。

readline は、read と同じように、ストリーム位置がファイルの終わりだった場合、空文字列を戻り値として返します。

次のプログラムの中で定義されている quote という関数は、パス名を受け取って、それによって指定されるテキストファイルの内容を構成しているそれぞれの行の先頭に大なり (>) と空白を挿入した結果を出力します。

プログラムの例 quote.py

```
def quote(filename):
    f = open(filename)
    eof = False
    while not eof:
        line = f.readline()
        if line == '':
            eof = True
        else:
            print('> ' + line, end='')
    f.close()
```

実行例

```
>>> from quote import *
>>> quote('puellae.txt')
> Kaname Madoka
> Akemi Homura
> Miki Sayaka
> Tomoe Mami
```

9.2.4 for 文による読み込みの繰り返し

ファイルオブジェクトは、繰り返し可能オブジェクトです。ということは、for 文を使うことによって、それに対する繰り返しを実行することができるということです。

for 文を使ってファイルオブジェクトに対する繰り返しを実行すると、繰り返しのたびに、ファイルオブジェクトから 1 個の行が読み込まれて、識別子がそれに束縛されます。

次のプログラムの中で定義されている line_number という関数は、パス名を受け取って、それによって指定されるテキストファイルの内容を構成しているそれぞれの行の先頭に行番号を挿入した結果を出力します。

プログラムの例 linenumber.py

```
def line_number(filename):
    n = 0
    f = open(filename)
    for line in f:
        n += 1
        print(str(n) + ': ' + line, end='')
    f.close()
```

実行例

```
>>> from linenumber import *
>>> line_number('puellae.txt')
1: Kaname Madoka
2: Akemi Homura
3: Miki Sayaka
4: Tomoe Mami
```

9.3 ファイルへの書き込み

9.3.1 ファイルからデータを読み込むメソッド

ファイルオブジェクトは、write というメソッドを持っています。このメソッドは、引数として受け取った文字列をストリーム位置に書き込んで、ストリーム位置をその直後へ移動させます。

次のプログラムの中で定義されている write_file という関数は、1 個目の引数としてパス名、2 個目の引数として文字列を受け取って、そのパス名で指定されたテキストファイルに、その文字列と改行を書き込みます。

プログラムの例 writefile.py

```
def write_file(filename, s):
    f = open(filename, 'w')
    f.write(s + '\n')
    f.close()
```

実行例

```
>>> from writefile import *
>>> write_file('shell.txt', 'Kusanagi Motoko')
```

書き込んだファイル shell.txt

```
Kusanagi Motoko
```

9.3.2 print によるファイルへの書き込み

ファイルオブジェクトが持っているメソッドではなくて、print を使うことによって文字列をファイルへ書き込むことも可能です。

print を使ってファイルに文字列を書き込みたいときは、そのファイルをオープンすることによって得られたファイルオブジェクトを、file というキーワード引数で print に渡します。

次のプログラムの中で定義されている print_file という関数は、1 個目の引数としてパス名、2 個目の引数として文字列を受け取って、そのパス名で指定されたテキストファイルに、その文字列と改行を書き込みます。

プログラムの例 printfile.py

```
def print_file(filename, s):
    f = open(filename, 'w')
    print(s, file=f)
    f.close()
```

実行例

```
>>> from printfile import *
>>> print_file('eden.txt', 'Morimi Saki')
```

書き込んだファイル eden.txt

```
Morimi Saki
```

9.4 モジュール

9.4.1 モジュールの基礎

処理系の中に取り込んで使うことのできる、処理系の外部にある機能は、「ライブラリー」(library) と呼ばれます。そして、処理系とともに配布されるライブラリーは、「標準ライブラリー」(standard library) と呼ばれます。

Python のライブラリーは、「モジュール」(module) と呼ばれるものから構成されています。処理系の中にモジュールを取り込むことを、モジュールを「インポートする」(import) と言います。

モジュールをインポートしたいときは、次の 2 種類の文のうちのどちらかを書きます。

- from 文 (from statement)
- import 文 (import statement)

9.4.2 from 文

「from 文」(import statement) と呼ばれる文を書くことによって、モジュールをインポートすることができます。

第 1.3.6 項で、インタラクティブシェルを使ってファイルに保存されているプログラムをインタプリタに実行させたいときは、

```
from モジュール名 import *
```

という文をインタラクティブシェルに入力すればいい、と説明しましたが、この文が from 文です。つまり、Python では、ファイルに保存されているプログラムというのは、すべてモジュールとして扱われるのです。

標準ライブラリーの中にある math というモジュールは、数学で使われるさまざまな浮動小数点数や関数を定義しています。たとえば、このモジュールは、 pi という識別子を円周率に束縛しています。

```
>>> from math import *
>>> pi
3.141592653589793
```

from 文をプログラムの先頭に書いておくと、そのプログラムの中で、from 文によってインポートされたものを利用することができるようになります。

次のプログラムの中で定義されている areaOfCircle という関数は、円の半径を受け取って、その円の面積を返します。

プログラムの例 areaofcircle.py

```
from math import *

def areaOfCircle(radius): return pi * radius * radius
```

実行例

```
>>> from areaofcircle import *
>>> areaOfCircle(5)
78.53981633974483
```

from 文の末尾には、これまで、アスタリスク (*) を書いてきました。このアスタリスクは、モジュールの中で定義されているすべてのオブジェクトをインポートするという意味です。すべてのオブジェクトではなくて、特定のオブジェクトだけをインポートしたいときは、アスタリスクの代わりに、インポートしたいオブジェクトの名前をコンマ (,) で区切って並べたものを書きます。たとえば、

```
from math import sin, cos, tan
```

という from 文は、 math の中で定義されているオブジェクトのうちの、 sin 、 cos 、 tan という三つの関数だけをインポートします。

9.4.3 import 文

複数のモジュールをインポートすると、しばしば、それぞれのモジュールの中で、同じ識別子が異なるものに束縛されているために、名前の競合が起きてしまうことがあります。

名前の競合は、モジュールを修飾付きでインポートすることによって避けることができます。モジュールを修飾付きでインポートしたいときは、from 文ではなくて、「import 文」(import statement) と呼ばれる文を書きます。

import 文は、

```
import モジュール名
```

と書きます。そうすると、「モジュール名」のところに書かれた名前のモジュールが修飾付きでインポートされます。

モジュールを修飾付きでインポートした場合、そのモジュールの中で定義されたものは、識別子をモジュール名で修飾した名前を使わなければ指定することができません。名前にモジュール名が含まれるということは、たとえ別のモジュールで同じ識別子が使われていたとしても、名前の競合が起きる心配はないということです。

識別子をモジュール名で修飾した名前は、

```
モジュール名 . 識別子
```

と書きます。たとえば、

```
import math
```

という import 文で math をインポートした場合、 pi を使うためには、

```
math.pi
```

という名前を書く必要があります。

```
>>> import math
>>> math.pi
3.141592653589793
```

9.5 ファイルに対するその他の処理

9.5.1 この節について

標準ライブラリーの中にある os というモジュールと os.path というモジュールは、ファイルに対する処理を実行するさまざまな関数を定義しています。

この節では、それらのモジュールの中で定義されている関数のうちのいくつかを紹介したいと思います。

9.5.2 os で定義されている関数

まず、os というモジュールの中で定義されている関数のうちのいくつかを紹介しましょう。

os の中で定義されている関数としては、次のようなものがあります。

mkdir 引数としてパス名を受け取って、そのパス名で指定されたディレクトリを作成する。

rmdir 引数としてパス名を受け取って、そのパス名で指定されたディレクトリを削除する。ただし、そのディレクトリは空でないといけない。

remove 引数としてパス名を受け取って、そのパス名で指定されたファイルを削除する。ディレクトリを削除することはできない。

rename 引数として 2 個のパス名を受け取って、1 個目のパス名で指定されたファイルまたはディレクトリの名前を 2 個目のパス名に変更する。

listdir 引数としてパス名を受け取って、そのパス名で指定されたディレクトリの一覧（ディレクトリの中にあるものの名前から構成されるリスト）を戻り値として返す。引数を省略した場合は、カレントディレクトリの一覧を返す。

stat 引数としてパス名を受け取って、そのパス名で指定されたファイルまたはディレクトリの情報を持つオブジェクトを返す。戻り値のオブジェクトは、ファイルまたはディレクトリの情報に束縛される次のような属性を持っている。

st_size 大きさ（バイト数）。

st_mtime　最終アクセス時刻。単位は秒、クラスは float。
st_mtime_ns　最終アクセス時刻。単位はミリ秒、クラスは int。
st_mtime　最終更新時刻。単位は秒、クラスは float。
st_mtime_ns　最終更新時刻。単位はミリ秒、クラスは int。

utime　引数としてパス名を受け取って、そのパス名で指定されたファイルに対して、最終アクセス時刻と最終更新時刻を設定する。最終アクセス時刻と最終更新時刻は、times または ns というキーワード引数で、

(最終アクセス時刻, 最終更新時刻)

というタプルの形で渡す。単位が秒の場合は times、単位がミリ秒の場合は ns を使う。どちらのキーワード引数でも時刻が渡されなかった場合は現在の時刻を設定する。

9.5.3 os.path で定義されている関数

次に、os.path というモジュールの中で定義されている関数のうちのいくつかを紹介しましょう。

os.path の中で定義されている関数としては、次のようなものがあります。

basename　引数としてパス名を受け取って、そのパス名の末尾の名前を返す。
dirname　引数としてパス名を受け取って、そのパス名から末尾の名前を取り除いた残りの部分を返す。
split　引数としてパス名を受け取って、そのパス名から末尾の名前を取り除いた残りの部分と、末尾の名前に分解して、それらから構成されるタプルを返す。
join　引数として何個かの名前を受け取って、それらの名前を結合することによってできるパス名を返す。
exists　引数としてパス名を受け取って、そのパス名で指定されたものが実在するならば True、そうでなければ偽を返す。
isfile　引数としてパス名を受け取って、そのパス名で指定されたものが実在するファイルならば True、そうでなければ偽を返す。
isdir　引数としてパス名を受け取って、そのパス名で指定されたものが実在するディレクトリならば True、そうでなければ偽を返す。

次のプログラムの中で定義されている sizelist という関数は、ディレクトリのパス名を受け取って、そのディレクトリの中にあるすべてのファイルについて、そのファイルの名前と大きさを出力します。

プログラムの例　sizelist.py

```
from os import *
from os.path import *

def sizelist(dirname):
    for item in listdir(dirname):
        name = join(dirname, item)
        if isfile(name):
            st = stat(name)
            print(item + ': ' + str(st.st_size))
```

実行例

```
>>> from sizelist import *
>>> sizelist('/home/hoge')
umiushi.txt: 370
namako.txt: 603
kamenote.txt: 228
```

第 10 章　GUI の基礎

10.1　GUI の基礎の基礎

10.1.1　GUI とは何か

機械などが持っている、人間に何かを報告したり人間からの指示を受け付けたりする部分のことを、「ユーザーインターフェース」(user interface) と言います。

コンピュータのプログラムも、たいていのものは何らかのユーザーインターフェースを持っています。プログラムのユーザーインターフェースとしては、現在、CUI と GUI と呼ばれる 2 種類のものが主流です。

CUI(character user interface) というのは、文字列を媒介とするユーザーインターフェースのことです。CUI の場合、プログラムは文字列を出力することによって人間に何かを報告し、人間はキーボードから文字列を入力することによってプログラムに指示を与えます。

それに対して、GUI(graphical user interface) は、図形を媒介とするユーザーインターフェースです。GUI の場合、プログラムはモニターの画面に図形を表示することによって人間に何かを報告し、人間はマウスなどの装置を使ってその図形を操作することによってプログラムに指示を与えます。

10.1.2　Tk

Python は、その処理系にも、その標準ライブラリーにも、GUI を作る機能はありません。ですから、GUI を持つプログラムを Python で書くためには、Python の外部にある、GUI を作る機能を利用する必要があります。

GUI を持つプログラムを Python で書く場合に利用されるのは、Tk と呼ばれるものです。これは、John Ousterhout さんという人が作った GUI のライブラリーで、Tk という名前は、toolkit という単語を縮めたものです。

Tk は、もともとは Tcl というプログラミング言語で使うために作られたものですが、現在では Tcl だけではなくて、Perl や Ruby や Python でもそれを使うことができます。

10.1.3　tkinter

Python のプログラムが Tk の機能を利用するためには、tkinter と呼ばれる、そのためのインターフェースが必要になります。

tkinter は、標準ライブラリーに含まれている `tkinter` という名前のモジュールの中で、さまざまなクラスとして定義されています。ですから、Python のプログラムは、このモジュールをインポートして、その中で定義されているクラスを使うことによって、Tk を利用することができます。

10.2　ウィジェット

10.2.1　ウィジェットの基礎

GUI は、画面の上に表示されるさまざまな部品から構成されます。そのような、GUI を作るための部品は、「ウィジェット」(widget) と呼ばれます。ちなみに、widget という単語は、window gadget という言葉を縮めたものだと言われています。ウィジェットには、ボタン、ラベル、キャンバス、エントリー、リストボックス、フレームなどのさまざまな種類があります。

tkinter では、ウィジェットは、「ウィジェットクラス」(widget class) と呼ばれるクラスから生成されたオブジェクトによって表示されます。たとえば、ラベルという種類のウィジェットは、`Label` というクラスから生成されたオブジェクトによって表示されます。なお、「ウィジェット」という言葉は、画面の上に表示される部品という意味だけではなくて、それを表示しているオブジェクトという意味でも使われます。

10.2.2　ルートウィジェット

GUI を持つプログラムは、通常、1 個以上のウィンドウを表示します。tkinter では、それらのウィンドウのうちでもっとも中心となるウィンドウを、「ルートウィジェット」(root widget) と呼びます。

tkinter では、ルートウィジェットは、`Tk` というウィジェットクラスから生成されます。

10.2.3 イベントループ

GUI を持つプログラムは、人間からの指示を受け取って、その指示に対応する処理を実行する、ということを何度も繰り返します。そのような繰り返しは、「イベントループ」(event loop) と呼ばれます。

tkinter では、ルートウィジェットが持っている `mainloop` というメソッドがイベントループになっています。tkinter を使って GUI を作るプログラムは、ウィジェットを生成したのちに、かならずこのメソッドを呼び出す必要があります（引数は不要です）。

次のプログラムは、ルートウィジェットを生成して、イベントループを実行します。

プログラムの例 `tk.py`

```
from tkinter import *
root = Tk()
root.mainloop()
```

10.2.4 ウィンドウのタイトル

ウィンドウに対して、そのタイトルバーに表示するタイトルを設定したいときは、ウィンドウが持っている `title` というメソッドを使います。このメソッドは、引数として文字列を受け取って、その文字列をタイトルとして自身に設定します。

次のプログラムは、「私はこのウィンドウのタイトルです。」というタイトルがタイトルバーに表示されたウィンドウを表示します。

プログラムの例 `title.py`

```
from tkinter import *
root = Tk()
root.title('私はこのウィンドウのタイトルです。')
root.mainloop()
```

10.2.5 ラベル

文字列（テキスト）をその上に表示することを目的として使われるウィジェットは、「ラベル」(label) と呼ばれます。ラベルは、`Label` というウィジェットクラスから生成されます。

ルートウィジェット以外のウィジェットを生成するときには、そのウィジェットを生成するクラスに対して、そのウィジェットの親となるウィジェットを引数として渡す必要があります。たとえば、`root` という識別子に束縛されているルートウィジェットを親としてラベルを生成したいという場合は、

```
Label(root)
```

という呼び出しを書く必要があります。

ウィジェットクラスに対しては、さまざまなキーワード引数を渡すことができます。ウィジェットクラスは、受け取ったキーワード引数を、生成するウィジェットのさまざまな状態を設定するために使います。

`text` というキーワード引数で文字列を `Label` に渡すと、その文字列がラベルの上に表示されたラベルが生成されます。たとえば、

```
Label(root, text='私はラベルです。')
```

という呼び出しで `Label` を呼び出すと、「私はラベルです。」という文字列が表示されたラベルが生成されます。

10.2.6 ジオメトリーマネージャー

ウィンドウ以外のウィジェットは、単独では画面の上に表示されません。それを画面に表示するためには、それの親になっているウィジェットの上にそれを取り付ける必要があります。

親のウィジェットの上にウィジェットを取り付けたいときは、「ジオメトリーマネージャー」(geometry manager) と呼ばれるものを使います。ジオメトリーマネージャーというのは、ウィジェットが持っているメソッドで、自身の配置を管理するという動作をするもののことです。

tkinter には、`pack`、`grid`、`place` という 3 種類のジオメトリーマネージャーがあります。それらのうちで、もっともよく使われるのは、`pack` というジオメトリーマネージャーです。`pack` は、親のウィジェットを必要最小限に小さくして、その中に自身を詰め込む、という動作をします。

次のプログラムは、ルートウィジェットを親としてラベルを生成して、そのラベルを pack を使って親の上に配置します。

プログラムの例　label.py

```
from tkinter import *
root = Tk()
label = Label(root, text=' 私はラベルです。')
label.pack()
root.mainloop()
```

ジオメトリーマネージャーについては、第 14 章で、もう少し詳しく説明したいと思います。

10.3　フォント

10.3.1　フォント記述子

ウィジェットの上に文字を表示するために使うフォントは、「フォント記述子」(font descriptor) と呼ばれる文字列を書くによって、自由に指定することができます。

フォント記述子の中には、最低限、Times、Helvetica、Courier というような、「フォントファミリー名」(font family name) と呼ばれる、フォントを識別するための名前を書く必要があります。

ウィジェットクラスに対して、font というキーワード引数でフォント記述子を渡すと、生成されたウィジェットは、それによって記述されたフォントを使って文字列を表示します。

次のプログラムは、Times、Helvetica、Courier というそれぞれのフォントを使って文字列を表示するラベルを生成します。

プログラムの例　family.py

```
from tkinter import *
root = Tk()
label1 = Label(root, text='Times', font='Times')
label1.pack()
label2 = Label(root, text='Helvetica', font='Helvetica')
label2.pack()
label3 = Label(root, text='Courier', font='Courier')
label3.pack()
root.mainloop()
```

10.3.2　フォントの大きさ

フォントの大きさを指定したいときは、

　フォントファミリー名 大きさ

という形のフォント記述子を書きます。大きさは、ポイントを単位とする数字で記述します。

次のプログラムは、30 ポイント、60 ポイント、120 ポイントというそれぞれの大きさのフォントを使って文字列を表示するラベルを生成します。

プログラムの例　size.py

```
from tkinter import *
root = Tk()
label1 = Label(root, text='30', font='Times 30')
label1.pack()
label2 = Label(root, text='60', font='Times 60')
label2.pack()
label3 = Label(root, text='120', font='Times 120')
label3.pack()
root.mainloop()
```

10.3.3　スタイル

フォントを太字にするとか、イタリック体にするという、フォントのスタイルを指定したいときは、

　フォントファミリー名 大きさ スタイル

という形のフォント記述子を書きます。「スタイル」というところに書くことができるのは、次の 3 種類です。

`bold`	太字。
`italic`	イタリック体。
`bold italic`	太字でかつイタリック体。

次のプログラムは、太字、イタリック体、太字でかつイタリック体、というそれぞれのフォントを使って文字列を表示するラベルを生成します。

プログラムの例 `style.py`

```
from tkinter import *
root = Tk()
label1 = Label(root, text='bold', font='Times 60 bold')
label1.pack()
label2 = Label(root, text='italic', font='Times 60 italic')
label2.pack()
label3 = Label(root, text='bold italic',
    font='Times 60 bold italic')
label3.pack()
root.mainloop()
```

10.4 色

10.4.1 色をあらわす文字列

ウィジェットの前景色（文字の色）や背景色は、色をあらわす文字列を書くことによって指定することができます。

色は、光の三原色のそれぞれの明るさを、

#*rgb*

#*rrggbb*

#*rrrgggbbb*

という形で並べることによって記述されます。*r*、*rr*、*rrr* のところには赤の比率、*g*、*gg*、*ggg* のところには緑の比率、*b*、*bb*、*bbb* のところには青の比率を、1 桁から 3 桁までの 16 進数で記述します。たとえば、`#f00` と書けば赤色、`#0ff` と書けば水色、`#f80` と書けばオレンジ色をあらわすことになります。

10.4.2 前景色

ウィジェットクラスに対して、`foreground` というキーワード引数で色の記述を渡すと、その色を前景色（文字の色）とするウィジェットが生成されます。

次のプログラムは、赤、ネイビー、オリーブというそれぞれの色を前景色とするラベルを生成します。

プログラムの例 `foreground.py`

```
from tkinter import *
root = Tk()
label1 = Label(root, text='red', font='Times 60',
    foreground='#f00')
label1.pack()
label2 = Label(root, text='navy', font='Times 60',
    foreground='#008')
label2.pack()
label3 = Label(root, text='olive', font='Times 60',
    foreground='#880')
label3.pack()
root.mainloop()
```

10.4.3　背景色

ウィジェットクラスに対して、background というキーワード引数で色の記述を渡すと、その色を背景色とするウィジェットが生成されます。

次のプログラムは、水色、黄色、黒というそれぞれの色を背景色とするラベルを生成します。

プログラムの例　background.py

```
from tkinter import *
root = Tk()
label1 = Label(root, text='aqua', font='Times 60',
    background='#0ff')
label1.pack()
label2 = Label(root, text='yellow', font='Times 60',
    background='#ff0')
label2.pack()
label3 = Label(root, text='black', font='Times 60',
    foreground='#fff', background='#000')
label3.pack()
root.mainloop()
```

第 11 章　イベント

11.1　イベントの基礎

11.1.1　バインディング

GUI を持つプログラムは、それに対して人間が何らかの操作をしたときに、それに対応する処理を実行します。GUI の上で何らかの操作が発生した、という出来事は、「イベント」(event) と呼ばれます。

イベントが発生したときに実行される処理は、「イベント処理」(event processing) と呼ばれます。そして、イベント処理を実行する関数やメソッドは、「イベントハンドラー」(event handler) と呼ばれます。

イベントに対してイベントハンドラーを割り当てることを、イベントにイベントハンドラーを「バインドする」(bind) と言います（名詞形は「バインディング」(binding) です）。

ウィジェットは、bind というメソッドを持っています。このメソッドは、自身の上で発生するイベントに対してイベントハンドラーをバインドします。ウィジェットは、イベントが自分の上で発生した場合、bind によってバインドされているイベントハンドラーを呼び出します。

11.1.2　イベントオブジェクト

ウィジェットは、イベントハンドラーを呼び出すときに、そのイベントハンドラーに 1 個の引数を渡します。ですから、イベントハンドラーは、1 個の引数を受け取るように定義しておく必要があります。

ウィジェットが引数としてイベントハンドラーに渡すオブジェクトは、「イベントオブジェクト」(event object) と呼ばれます。イベントオブジェクトは、どのようなイベントが発生したのかという情報を持っています。

11.1.3　イベントパターン

bind は、2 個の引数を受け取ります。1 個目の引数は、「イベントパターン」(event pattern) と呼ばれる、イベントの種類を指定する文字列です。そして 2 個目の引数は、イベントハンドラーです。

イベントパターンは、

<修飾子-タイプ-詳細>

という形式で書きます。ただし、最低限必要なのは「タイプ」という部分だけです。「タイプ」の部分には、「イベントタイプ」(event type) と呼ばれる、イベントの概略を示す文字列を書きます。たとえば、Button という文字列は、「マウスのボタンが押された」というイベントを示すイベントタイプで、ButtonRelease という文字列は、「マウスのボタンが離された」というイベントを示すイベントタイプです。

次のプログラムが表示するウィンドウは、マウスポインターが自身の上にあるとき、マウスのボタンが押されると、「マウスのボタンが押されました。」という文字列を出力して、マウスのボタンが離されると、「マウスのボタンが離されました。」という文字列を出力します。

プログラムの例 bind.py

```
from tkinter import *

def pressed(event):
    print(' マウスのボタンが押されました。')

def released(event):
    print(' マウスのボタンが離されました。')

root = Tk()
root.bind('<Button>', pressed)
root.bind('<ButtonRelease>', released)
root.mainloop()
```

11.1.4 ウィジェットの状態の変更

第 10.2.5 項で説明したように、ウィジェットクラスはさまざまなキーワード引数を受け取ることができて、それらのキーワード引数は、ウィジェットの状態を設定するために使われます。

ウィジェットの状態は、生成されたのちも、自由に変更することが可能です。ウィジェットの状態を変更したいときは、そのウィジェットが持っている config というメソッドを呼び出します。ウィジェットクラスに渡すものと同じキーワード引数を config に渡すと、それによってウィジェットの状態が変更されます。

次のプログラムが表示するラベルは、マウスポインターが自身の上にあるときにマウスのボタンが押されると色を反転させて、ボタンが離されると、色を元に戻して、表示している整数を 1 だけ増加させます。

プログラムの例 config.py

```
from tkinter import *

def pressed(event):
    label.config(foreground='#0ff', background='#008')

def released(event):
    global count
    count += 1
    label.config(text=str(count), foreground='#008',
        background='#0ff')

count = 0
root = Tk()
label = Label(root, text='0', font='Times 180',
    foreground='#008', background='#0ff')
label.bind('<Button>', pressed)
label.bind('<ButtonRelease>', released)
label.pack()
root.mainloop()
```

11.2 マウスによるイベント

11.2.1 マウスのイベントタイプ

マウスによるイベントをバインドするためのイベントパターンは、表 11.1 のイベントタイプを使うことによって書くことができます。

11.2.2 マウスの特定のボタンに限定したバインディング

マウスの特定のボタンによるイベントに限定してイベントハンドラーをバインドしたいときは、

<タイプ-番号>

タイプ	説明
Button	マウスのボタンが押された。
ButtonPress	Button と同じ。
ButtonRelease	マウスのボタンが離された。
Motion	マウスが移動した。
Enter	マウスポインターが重なった。
Leave	マウスポインターが離れた。

表 11.1: マウスのイベントタイプ

という形式のイベントパターンを書きます。この中の「番号」というところには、マウスのボタンを指定する番号を書きます。

3 個のボタンを持つマウスの場合、それぞれのボタンには、左から順番に、1、2、3、という番号が与えられています。ボタンが 2 個しかないマウスの場合は、左が 1 で右が 3 です。

次のプログラムによって表示される数字は、マウスの左ボタンでクリックすると 1 だけ増加して、右ボタンでクリックすると 1 だけ減少します。

プログラムの例 button.py

```
from tkinter import *

def increase(event):
    global count
    count += 1
    label.config(text=str(count))

def decrease(event):
    global count
    count -= 1
    label.config(text=str(count))

count = 0
root = Tk()
label = Label(root, text='0', font='Times 180')
label.bind('<ButtonRelease-1>', increase)
label.bind('<ButtonRelease-3>', decrease)
label.pack()
root.mainloop()
```

11.2.3 マウスのイベント修飾子

第 11.1.3 項で説明したように、イベントパターンというのは、

<修飾子-タイプ-詳細>

という形式を持つ文字列のことです。この中の「修飾子」のところに書くことのできる文字列は、「イベント修飾子」(event modifier) と呼ばれます。イベント修飾子というのは、いくつかの操作の組み合わせであるようなイベントを指定したいときに書く文字列です。

Button1、Button2、Button3 という文字列のそれぞれは、マウスの左ボタン、真ん中ボタン、右ボタンが押されているときに発生するイベントを指定するイベント修飾子です。たとえば、

<Button1-Motion>

というイベントパターンを書くことによって、マウスの左ボタンでドラッグされるというイベントを指定することができます。

次のプログラムによって表示される数字は、マウスの左ボタンでドラッグすると、それに応じて増えていきます。

プログラムの例 drag.py

```
from tkinter import *

def increase(event):
```

```
    global count
    count += 1
    label.config(text=str(count))

count = 0
root = Tk()
label = Label(root, text='0', font='Times 180')
label.bind('<Button1-Motion>', increase)
label.pack()
root.mainloop()
```

11.2.4 マウスポインターの座標

第 11.1.2 項で説明したように、イベントハンドラーは、「イベントオブジェクト」と呼ばれるオブジェクトを引数として受け取ります。イベントオブジェクトは、発生したイベントについてのさまざまな情報を持っています。

イベントオブジェクトは、x と y という属性を持っています。これらの属性は、イベントが発生したときのマウスポインターの位置を示す座標に束縛されています。x は x 座標、y は y 座標です。

ウィジェットの座標系は、左上の隅に原点があって、x 軸は右向き、y 軸は下向き、そして単位はピクセルです。

次のプログラムが生成するラベルは、その上にマウスポインターがあるときにマウスを移動させると、そのときのマウスポインターの座標を表示します。

プログラムの例 coordinate.py

```
from tkinter import *

def show_coordinate(event):
    label.config(
        text = '(' + str(event.x) + ', ' + str(event.y) + ')')

root = Tk()
label = Label(root, text='(0, 0)', font='Times 100')
label.bind('<Motion>', show_coordinate)
label.pack()
root.mainloop()
```

11.3 キーボードによるイベント

11.3.1 キーボードのイベントタイプ

キーボードによるイベントをバインドするためのイベントパターンは、次のイベントタイプを使うことによって書くことができます。

Key	キーボードのキーが押された。
KeyPress	Key と同じ。
KeyRelease	キーボードのキーが離された。

次のプログラムは、キーボードのキーが押されると、「キーボードのキーが押されました。」という文字列を出力して、キーボードのキーが離されると、「キーボードのキーが離されました。」という文字列を出力します。

プログラムの例 key.py

```
from tkinter import *

def pressed(event):
    print(' キーボードのキーが押されました。')

def released(event):
    print(' キーボードのキーが離されました。')

root = Tk()
root.bind('<Key>', pressed)
```

```
root.bind('<KeyRelease>', released)
root.mainloop()
```

11.3.2 キーボードの特定のキーに限定したバインディング

キーボードの特定のキーによるイベントに限定してイベントハンドラーをバインドしたいときは、

<タイプ-キーシム>

という形式のイベントパターンを書きます。

「キーシム」(keysym) というのは、キーボードの上にあるそれぞれのキーを識別するための文字列のことです。たとえば、スペースは space、コンマは comma、スラッシュは slash、エンターは Enter、右のシフトは Shift_R、上向き矢印は Up、下向き矢印は Down、というキーシムによって識別されます。なお、数字と英字のキーについては、そのキーの文字がそのままキーシムになっています。

次のプログラムによって表示される数字は、上向き矢印キーを押すと 1 だけ増加して、下向き矢印キーを押すと 1 だけ減少します。

プログラムの例 updown.py

```
from tkinter import *

def increase(event):
    global count
    count += 1
    label.config(text=str(count))

def decrease(event):
    global count
    count -= 1
    label.config(text=str(count))

count = 0
root = Tk()
root.bind('<Key-Up>', increase)
root.bind('<Key-Down>', decrease)
label = Label(root, text='0', font='Times 180')
label.pack()
root.mainloop()
```

11.3.3 キーボードのイベント修飾子

Shift、Control、Alt という文字列のそれぞれは、シフトキー、コントロールキー、オルトキーが押されているときに発生するイベントを指定するイベント修飾子です。たとえば、

<Control-Key-space>

というイベントパターンを書くことによって、コントロールキーが押されているときにスペースキーが押されるというイベントを指定することができます。

次のプログラムは、コントロールキーが押されているときにスペースキーが押されると、「コントロール＋スペースが押されました。」という文字列を出力して、コントロールキーが押されているときにスペースキーが離されると、「コントロール＋スペースが離されました。」という文字列を出力します。

プログラムの例 control.py

```
from tkinter import *

def pressed(event):
    print('コントロール＋スペースが押されました。')

def released(event):
    print('コントロール＋スペースが離されました。')

root = Tk()
root.bind('<Control-Key-space>', pressed)
```

```
root.bind('<Control-KeyRelease-space>', released)
root.mainloop()
```

11.3.4　イベントを発生させたキーのキーシム

イベントオブジェクトは、keysym という属性を持っています。この属性は、キーボードによるイベントをあらわすイベントオブジェクトでは、イベントを発生させたキーのキーシムに束縛されています。

次のプログラムは、キーボードのキーが押されると、押されたキーのキーシムを出力します。

プログラムの例　keysym.py

```
from tkinter import *

def pressed(event):
    print('keysym = ' + event.keysym)

root = Tk()
root.bind('<Key>', pressed)
root.mainloop()
```

第 12 章　キャンバス

12.1　キャンバスの基礎

12.1.1　キャンバスとは何か

tkinter は、「キャンバス」(canvas) と呼ばれるウィジェットを扱う機能を持っています。

キャンバスは、その上にグラフィックスを描画するという目的で使われるウィジェットです。

12.1.2　キャンバスの生成

キャンバスは、Canvas というウィジェットクラスから生成されます。

Canvas を呼び出してキャンバスを生成する方法は、Label を呼び出してラベルを生成する方法と同じです。1 個目の引数としてウィジェットを渡して、さらにいくつかのキーワード引数を渡すと、1 個目のウィジェットを親とするキャンバスが生成されて、キーワード引数がそれの状態として設定されます。

Canvas には、通常、次のようなキーワード引数を渡します。

width	横の長さ。単位はピクセル。
height	縦の長さ。単位はピクセル。
background	背景色。

次のプログラムは、横の長さが 640 ピクセル、縦の長さが 480 ピクセルで、背景色がオレンジ色のキャンバスを生成します。

プログラムの例　canvas.py

```
from tkinter import *
root = Tk()
canvas = Canvas(root, width=640, height=480, background='#f80')
canvas.pack()
root.mainloop()
```

12.1.3　長方形

キャンバスはさまざまなメソッドを持っています。それらの中には、キャンバスの上にグラフィックスを描画するメソッドもあります。

たとえば、キャンバスが持っている create_rectangle というメソッドを呼び出すことによって、キャンバスの上に長方形を描画することができます。

長方形の位置と形状は、次の 4 個の引数を create_rectangle に渡すことによって指定します。

1 個目　左上の頂点の x 座標。

2 個目　左上の頂点の y 座標。
3 個目　右下の頂点の x 座標。
4 個目　右下の頂点の y 座標。

次のプログラムは、左上の頂点が $(80, 80)$ で右下の頂点が $(560, 400)$ の長方形を描画します。

プログラムの例　rectangle.py

```
from tkinter import *
root = Tk()
canvas = Canvas(root, width=640, height=480)
canvas.create_rectangle(80, 80, 560, 400)
canvas.pack()
root.mainloop()
```

長方形以外のグラフィックスをキャンバスの上に描画するメソッドについては、第 12.2 節で紹介することにしたいと思います。

12.1.4　描画のキーワード引数

キャンバスの上にグラフィックスを描画するメソッドの多くは、次のようなキーワード引数を受け取ります。

`width`　線の幅。
`outline`　線の色。
`fill`　塗りつぶしの色。

次のプログラムは、線の幅が 20 ピクセル、線の色がネイビー、塗りつぶしの色が水色の長方形を描画します。

プログラムの例　rectcolor.py

```
from tkinter import *
root = Tk()
canvas = Canvas(root, width=640, height=480)
canvas.create_rectangle(80, 80, 560, 400, width=20,
    outline='#008', fill='#0ff')
canvas.pack()
root.mainloop()
```

`outline` と `fill` で、色をあらわす文字列として空文字列を渡した場合、それは、透明な色が指定されたと解釈されます。ちなみに、`fill` は、デフォルトが透明な色です。

次のプログラムは、線が透明で、内部が赤紫で塗りつぶされた長方形を描画します。

プログラムの例　transparent.py

```
from tkinter import *
root = Tk()
canvas = Canvas(root, width=640, height=480)
canvas.create_rectangle(80, 80, 560, 400, width=20,
    outline='', fill='#f0f')
canvas.pack()
root.mainloop()
```

12.2　グラフィックスの描画

12.2.1　この節について

第 12.1.3 項で説明したように、キャンバスはさまざまなメソッドを持っていて、それらの中には、キャンバスの上にグラフィックスを描画するものもあります。

この節では、キャンバスの上にグラフィックスを描画するメソッドを紹介したいと思います。ただし、長方形を描画するメソッドについては、すでに第 12.1.3 項で紹介しましたので、そちらを参照してください。

12.2.2 楕円

キャンバスは、create_oval というメソッドを持っています。このメソッドは、キャンバスの上に楕円を描画します。

create_oval は、楕円に外接する長方形の位置と形状を示す次の 4 個の引数を受け取ります。

1 個目　外接する長方形の左上の頂点の x 座標。
2 個目　外接する長方形の左上の頂点の y 座標。
3 個目　外接する長方形の右下の頂点の x 座標。
4 個目　外接する長方形の右下の頂点の y 座標。

次のプログラムは、外接する長方形の左上の頂点が $(80, 80)$ で右下の頂点が $(560, 400)$ の楕円を描画します。

プログラムの例　oval.py

```
from tkinter import *
root = Tk()
canvas = Canvas(root, width=640, height=480)
canvas.create_oval(80, 80, 560, 400, width=10,
    outline='#090', fill='#6f6')
canvas.pack()
root.mainloop()
```

12.2.3 円弧

キャンバスは、create_arc というメソッドを持っています。このメソッドは、キャンバスの上に円弧を描画します。

create_arc は、create_oval と同じように、円弧に外接する長方形の位置と形状を示す 4 個の引数を受け取ります。ただし、円弧の場合、それらの引数だけでは形状を完全に指定することができませんので、それらに加えて、さらに次のキーワード引数を渡す必要があります。

start　円弧の開始点の角度（単位は度）。角度は、中心から見て右方向が 0 度で、反時計回りに大きくなる。

extent　円弧の開始点から終了点までの角度（単位は度）。プラスの数値が指定された場合は開始点から終了点に向かって反時計回りに円弧を描画し、マイナスの数値が指定された場合は時計回りに円弧を描画する。

style　円弧のスタイル。次の識別子で指定する。

- ARC　円弧のみ。fill で渡された色は無視する。
- CHORD　開始点と終了点をつなぐ直線を追加する。
- PIESLICE　中心と開始点、中心と終了点をつなぐ直線を追加する。

次のプログラムは、4 個の円弧を描画します。スタイルは、上の二つが ARC、左下が CHORD、右下が PIESLICE です。

プログラムの例　arc.py

```
from tkinter import *
root = Tk()
canvas = Canvas(root, width=640, height=480)
canvas.create_arc(60, 50, 310, 220, start=45, extent=270,
    style=ARC, width=10, outline='#900', fill='#f66')
canvas.create_arc(330, 50, 580, 220, start=-225, extent=-270,
    style=ARC, width=10, outline='#900', fill='#f66')
canvas.create_arc(60, 260, 310, 430, start=45, extent=270,
    style=CHORD, width=10, outline='#900', fill='#f66')
canvas.create_arc(330, 260, 580, 430, start=-225, extent=-270,
    style=PIESLICE, width=10, outline='#900', fill='#f66')
canvas.pack()
root.mainloop()
```

12.2.4　折れ線

キャンバスは、create_line というメソッドを持っています。このメソッドは、キャンバスの上に折れ線を描画します。

create_line は、折れ線を構成する個々の頂点の座標を引数として受け取ります。たとえば、

$$x_1, y_1, x_2, y_2, \cdots, x_n, y_n$$

という引数を渡すと、create_line は、

$$(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$$

という座標の点を順番に直線で連結することによってできる折れ線を描画します。

多くのグラフィックスは、outline というキーワード引数で線の色を指定するわけですが、折れ線は、ほかのグラフィックスとは違って、outline ではなくて fill で線の色を指定します。

次のプログラムは、$(80, 80)$、$(80, 400)$、$(560, 80)$、$(560, 400)$ という座標の点を順番に直線で連結することによってできる折れ線を描画します。

プログラムの例　line.py

```
from tkinter import *
root = Tk()
canvas = Canvas(root, width=640, height=480)
canvas.create_line(80, 80, 80, 400, 560, 80, 560, 400,
    width=10, fill='#060')
canvas.pack()
root.mainloop()
```

12.2.5　多角形

キャンバスは、create_polygon というメソッドを持っています。このメソッドは、キャンバスの上に多角形を描画します。

create_polygon は、create_line と同じように、多角形を構成する個々の頂点の座標を引数として受け取ります。たとえば、

$$x_1, y_1, x_2, y_2, \cdots, x_n, y_n$$

という引数を渡すと、create_polygon は、

$$(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$$

という座標の点を順番に直線で連結して、さらに (x_n, y_n) と (x_1, y_1) を直線で連結することによってできる多角形を描画します。

次のプログラムは、$(80, 80)$、$(80, 400)$、$(560, 80)$、$(560, 400)$ という座標の点を順番に直線で連結して、さらに $(560, 400)$ と $(80, 80)$ を直線で連結することによってできる多角形を描画します。

プログラムの例　polygon.py

```
from tkinter import *
root = Tk()
canvas = Canvas(root, width=640, height=480)
canvas.create_polygon(80, 80, 80, 400, 560, 80, 560, 400,
    width=10, outline='#009', fill='#66f')
canvas.pack()
root.mainloop()
```

12.2.6　文字列

キャンバスは、create_text というメソッドを持っています。このメソッドは、キャンバスの上に文字列を描画します。

create_text は、文字列を描画する位置の座標を引数として受け取ります。文字列は、座標で指定された点を中心とする位置に描画されます。

さらに create_text は、次のようなキーワード引数を受け取ります。

text　描画する文字列。

font　文字列を描画するために使うフォント。

fill　文字列の色。

次のプログラムは、「私は文字列です。」という文字列を描画します。

プログラムの例 text.py

```
from tkinter import *
root = Tk()
canvas = Canvas(root, width=640, height=480)
canvas.create_text(320, 240, text='私は文字列です。',
    font='Times 40', fill='#600')
canvas.pack()
root.mainloop()
```

12.2.7 ウィジェット

キャンバスは、create_windowというメソッドを持っています。このメソッドは、キャンバスの上にウィジェットを描画します。ただし、描画するウィジェットは、描画に先立ってあらかじめ生成しておく必要があります。

create_windowは、ウィジェットを描画する位置の座標を引数として受け取ります。ウィジェットは、座標で指定された点を中心とする位置に描画されます。

create_windowは、描画するウィジェットを、window というキーワード引数で受け取ります。

次のプログラムは、ラベルをキャンバスの上に描画します。

プログラムの例 widget.py

```
from tkinter import *
root = Tk()
canvas = Canvas(root, width=640, height=480)
label = Label(root, text='私はラベルです。', font='Times 40',
    foreground='#066', background='#6ff')
canvas.create_window(320, 240, window=label)
canvas.pack()
root.mainloop()
```

12.3 グラフィックスの操作

12.3.1 グラフィックスの ID

キャンバスの上にグラフィックスを描画するメソッドは、戻り値として整数を返します。この整数は、描画されたグラフィックスを識別することのできる ID です。キャンバスの上に描画されたグラフィックスに対しては、その ID を指定することによって、さまざまな操作をすることができます。

12.3.2 グラフィックスのバインディング

キャンバスは、tag_bindというメソッドを持っています。このメソッドは、キャンバスの上に描画されたグラフィックスの上で発生したイベントに対してイベントハンドラーをバインドします。

tag_bindは、次の 3 個の引数を受け取ります。

1 個目　イベントハンドラーをバインドするグラフィックスの ID。

2 個目　イベントの種類を指定するイベントパターン。

3 個目　グラフィックスにバインドするイベントハンドラー。

次のプログラムは、2 個の楕円を描画します。そして、左の楕円がクリックされると、「左の楕円がクリックされました。」と出力して、右の楕円がクリックされると、「右の楕円がクリックされました。」と出力します。

プログラムの例 bindoval.py

```
from tkinter import *

def left_released(event):
    print('左の楕円がクリックされました。')

def right_released(event):
```

```
    print(' 右の楕円がクリックされました。')

root = Tk()
canvas = Canvas(root, width=640, height=480)
oval1 = canvas.create_oval(80, 80, 300, 400, width=10,
    outline='#900', fill='#f66')
canvas.tag_bind(oval1, '<ButtonRelease>', left_released)
oval2 = canvas.create_oval(340, 80, 560, 400, width=10,
    outline='#009', fill='#66f')
canvas.tag_bind(oval2, '<ButtonRelease>', right_released)
canvas.pack()
root.mainloop()
```

次のプログラムは、12 個の楕円を描画します。そして、楕円のどれかがクリックされると、クリックされた楕円の ID を出力します。

プログラムの例　bindoval2.py

```
from tkinter import *

class Oval:
    def __init__(self, x):
        self.id = canvas.create_oval(x, 120, x+30, 360,
            width=10, outline='#0f9', fill='#9fc')
        canvas.tag_bind(self.id, '<ButtonRelease>',
            self.released)
    def released(self, event):
        print(' 私の ID は' + str(self.id) + ' です。')

root = Tk()
canvas = Canvas(root, width=640, height=480)
for x in range(80, 560, 40):
    Oval(x)
canvas.pack()
root.mainloop()
```

12.3.3　グラフィックスの消去

キャンバスは、delete というメソッドを持っています。このメソッドは、キャンバスの上に描画されたグラフィックスを消去します。

delete は、消去するグラフィックスの ID を引数として受け取ります。

次のプログラムは、多数の楕円を描画します。そして、楕円のどれかがクリックされると、クリックされた楕円を消去します。

プログラムの例　delete.py

```
from tkinter import *

class Oval:
    def __init__(self, x1, y1):
        self.id = canvas.create_oval(x1, y1, x1+20, y1+20,
            outline='', fill='#009')
        canvas.tag_bind(self.id, '<ButtonRelease>',
            self.released)
    def released(self, event):
        canvas.delete(self.id)

root = Tk()
canvas = Canvas(root, width=640, height=480)
for x in range(80, 560, 20):
    for y in range(80, 380, 20):
        Oval(x, y)
canvas.pack()
root.mainloop()
```

12.3.4 グラフィックスの状態の変更

キャンバスは、itemconfigure というメソッドを持っています。このメソッドは、キャンバスの上に描画されたグラフィックスの状態を変更します。

itemconfigure は、状態を変更するグラフィックスの ID を 1 個目の引数として受け取って、さらに、状態をどのように変更するのかということを意味するキーワード引数を受け取ります。渡すことのできるキーワード引数は、グラフィックスを描画するメソッドに渡すことのできるキーワード引数と同じです。

次のプログラムが表示するキャンバスは、マウスポインターが自身の上にあるときにマウスのボタンが押されると、その上に描画されている整数の色を赤に変更して、ボタンが離されると、整数を 1 だけ増加させて、その色を元に戻します。

プログラムの例 itemconfigure.py

```
from tkinter import *

def pressed(event):
    canvas.itemconfigure(number, text=str(count), fill='#f00')

def released(event):
    global count
    count += 1
    canvas.itemconfigure(number, text=str(count), fill='#009')

count = 0
root = Tk()
canvas = Canvas(root, width=640, height=480)
canvas.bind('<Button>', pressed)
canvas.bind('<ButtonRelease>', released)
number = canvas.create_text(320, 240, text='0',
    font='Times 120', fill='#009')
canvas.pack()
root.mainloop()
```

12.3.5 グラフィックスの移動

キャンバスは、move というメソッドを持っています。このメソッドは、キャンバスの上に描画されたグラフィックスを移動させます。

move は、次の 3 個の引数を受け取ります。

1 個目　移動させるグラフィックスの ID。

2 個目　x 軸方向の移動距離。

3 個目　y 軸方向の移動距離。

次のプログラムは、1 個の楕円を描画します。そして、キーボードの矢印キーが押されると、上下左右のそれぞれの方向へ楕円を移動させます。

プログラムの例 move.py

```
from tkinter import *

def move(event):
    if event.keysym == 'Up':
        canvas.move(oval, 0, -step)
    elif event.keysym == 'Down':
        canvas.move(oval, 0, step)
    elif event.keysym == 'Left':
        canvas.move(oval, -step, 0)
    elif event.keysym == 'Right':
        canvas.move(oval, step, 0)

step = 10
root = Tk()
root.bind('<Key>', move)
canvas = Canvas(root, width=640, height=480)
oval = canvas.create_oval(300, 220, 340, 260,
    outline='', fill='#090')
```

```
canvas.pack()
root.mainloop()
```

次のプログラムは、8 個の楕円を描画します。そして、楕円のどれかがドラッグされると、ドラッグされた楕円を移動させます。

プログラムの例　dragoval.py

```
from tkinter import *

class Oval:
    def __init__(self, x):
        self.id = canvas.create_oval(x, 220, x+40, 260,
            outline='', fill='#069')
        canvas.tag_bind(self.id, '<Button-1>',
            self.pressed)
        canvas.tag_bind(self.id, '<Button1-Motion>',
            self.dragging)
    def pressed(self, event):
        self.x = event.x
        self.y = event.y
    def dragging(self, event):
        canvas.move(self.id, event.x-self.x, event.y-self.y)
        self.x = event.x
        self.y = event.y

root = Tk()
canvas = Canvas(root, width=640, height=480)
for x in range(80, 560, 60):
    Oval(x)
canvas.pack()
root.mainloop()
```

12.3.6　タグ

キャンバスの上に描画されたグラフィックスは、ID によって識別することができるわけですが、ID だけではなくて、「タグ」(tag) と呼ばれる文字列によって識別することも可能です。

グラフィックスを描画するメソッドに、tags というキーワード引数で文字列を渡すと、その文字列は、そのメソッドによって描画されたグラフィックスを識別するタグになります。たとえば、

```
canvas.create_oval(50, 50, 200, 200, tags='ovals')
```

という呼び出しで楕円を描画したとすると、その楕円は、ovals というタグによって識別することができます。

キャンバスが持っている、グラフィックスを操作するメソッドは、1 個目の引数として、操作の対象となるグラフィックスの ID を受け取るわけですが、ID の代わりにタグを渡すこともできます。その場合は、そのタグで識別されるグラフィックスが操作の対象となります。たとえば、

```
canvas.move('ovals', 40, 30)
```

と書くことによって、ovals というタグによって識別されるグラフィックスを移動させることができます。

ID は、個々のグラフィックスごとに異なる整数が与えられます。しかし、タグは ID とは違って、個々のグラフィックスごとに異なっている必要はありません。複数のグラフィックスに対して同一の文字列をタグとして与えることもできます。

複数のグラフィックスに対して同一の文字列をタグとして与えると、そのタグが与えられたすべてのグラフィックスに対して一斉に同一の操作を加えることが可能になります。

次のプログラムは、8 個の楕円を描画します。そして、楕円のどれかがドラッグされると、すべての楕円を移動させます。

プログラムの例　tags.py

```
from tkinter import *

class Oval:
    def __init__(self, x):
```

```
        canvas.create_oval(x, 220, x+40, 260,
            outline='', fill='#096', tags='ovals')

def pressed(event):
    global x, y
    x = event.x
    y = event.y

def dragging(event):
    global x, y
    canvas.move('ovals', event.x-x, event.y-y)
    x = event.x
    y = event.y

x = 0
y = 0
root = Tk()
canvas = Canvas(root, width=640, height=480)
for x in range(80, 560, 60):
    Oval(x)
canvas.tag_bind('ovals', '<Button-1>', pressed)
canvas.tag_bind('ovals', '<Button1-Motion>', dragging)
canvas.pack()
root.mainloop()
```

12.4 アニメーション

12.4.1 アフター

時間の経過とともに変化するグラフィックスは、「アニメーション」(animation) と呼ばれます。

アニメーションを作るためには、一定の時間ごとに何らかの動作を実行する仕組みが必要です。tkinter では、「アフター」(after) と呼ばれる仕組みを使うことによって、一定の時間ごとに何らかの動作を実行することができます。

アフターを使うためには、それを設定する必要があります。アフターを設定したいときは、ウィジェットが持っている `after` という関数を使います。

`after` は、次の 2 個の引数を受け取ります。

1 個目　2 個目の引数として受け取った関数を実行する間隔。単位はミリ秒。

2 個目　関数。

次のプログラムは、キャンバスの上に描画された数値を 100 ミリ秒ごとにカウントアップさせます。

プログラムの例　`after.py`

```
from tkinter import *

def countup():
    global count
    count += 1
    canvas.itemconfigure(number, text=str(count))
    root.after(100, countup)

count = 0
root = Tk()
canvas = Canvas(root, width=640, height=480)
number = canvas.create_text(320, 240, text='0',
    font='Times 120', fill='#090')
canvas.pack()
countup()
root.mainloop()
```

12.4.2 移動のアニメーション

それでは、アフターを使って、グラフィックスを移動させるアニメーションを作ってみましょう。

グラフィックスを移動させるアニメーションを作ろうとすると、その多くの場合には、移動させるグラフィックスの現在位置を取得することが必要になります。グラフィックスの現在位置は、キャンバスが持っている bbox というメソッドを呼び出すことによって取得することができます。

bbox は、グラフィックスの ID を受け取って、そのグラフィックスを取り囲む長方形を示す、

(x_1, y_1, x_2, y_2)

という形のタプルを戻り値として返します。(x_1, y_1) というのが長方形の左上の頂点で、(x_2, y_2) というのが長方形の右下の頂点です。

次のプログラムは、キャンバスの上に描画された楕円を左右に移動させます。

プログラムの例　rightleft.py

```
from tkinter import *

def rightleft():
    global direction
    x = canvas.bbox(oval)[0]
    if x > 550:
        direction = 1 # left
    elif x < 50:
        direction = 0 # right
    if direction == 0:
        canvas.move(oval, step, 0)
    else:
        canvas.move(oval, -step, 0)
    root.after(10, rightleft)

direction = 0 # right
step = 5
root = Tk()
canvas = Canvas(root, width=640, height=480)
oval = canvas.create_oval(50, 220, 90, 260,
    outline='', fill='#690')
canvas.pack()
rightleft()
root.mainloop()
```

12.4.3　インタラクティブなアニメーション

次に、インタラクティブなアニメーション、つまり、ユーザーが操作することのできるアニメーションを作ってみましょう。

次のプログラムは、矢印のキーを押すことによって、楕円が移動する方向を変更することができます。

プログラムの例　direction.py

```
from tkinter import *

def change_direction(event):
    global direction
    if event.keysym == 'Up':
        direction = 0
    elif event.keysym == 'Down':
        direction = 1
    elif event.keysym == 'Left':
        direction = 2
    elif event.keysym == 'Right':
        direction = 3

def move_oval():
    if direction == 0: # up
        canvas.move(oval, 0, -step)
    elif direction == 1: # down
        canvas.move(oval, 0, step)
    elif direction == 2: # left
        canvas.move(oval, -step, 0)
    else:
```

```
        canvas.move(oval, step, 0)
    root.after(10, move_oval)

direction = 3 # right
step = 5
root = Tk()
root.bind('<Key>', change_direction)
canvas = Canvas(root, width=640, height=480)
oval = canvas.create_oval(50, 220, 90, 260,
    outline='', fill='#096')
canvas.pack()
move_oval()
root.mainloop()
```

第 13 章　ボタンとメニュー

13.1　ボタンの基礎

13.1.1　ボタンとは何か

クリックされたときに何らかの動作を実行したり、自身の状態を変更したりするウィジェットは、「ボタン」(button) と呼ばれます。

13.1.2　ボタンの分類

tkinter には、次のような 3 種類のボタンがあります。

- プッシュボタン (push button)
- チェックボタン (check box)
- ラジオボタン (radio button)

「ボタン」という言葉は、本来はこれらのウィジェットの総称ですが、プッシュボタンのことを単に「ボタン」と呼ぶこともあります。

チェックボタンは、「チェックボックス」と呼ばれることもあります。

13.2　プッシュボタン

13.2.1　プッシュボタンの基礎

クリックされたときに何らかの動作を実行するボタンは、「プッシュボタン」(push button) または単に「ボタン」と呼ばれます。

13.2.2　プッシュボタンの生成

プッシュボタンは、`Button` というウィジェットクラスから生成されます。

`Button` には、1 個目の引数として親のウィジェットを渡して、さらに次のようなキーワード引数を渡します。

`text`	プッシュボタンの上に表示する文字列。
`command`	プッシュボタンがクリックされたときに実行される関数またはメソッド。

次のプログラムが表示するプッシュボタンは、クリックされると、「ボタンがクリックされました。」と出力します。

プログラムの例　`button.py`

```
from tkinter import *

def clicked():
    print(' ボタンがクリックされました。')

root = Tk()
button = Button(root, text=' 私はボタンです。',
    command=clicked, font='Times 20')
```

```
button.pack()
root.mainloop()
```

13.3　チェックボタン

13.3.1　チェックボタンの基礎

オンの状態（チェックが入っている状態）とオフの状態（チェックが入っていない状態という二通りの状態を持っていて、クリックすることによってそれらの状態を切り替えることのできるボタンは、「チェックボタン」(check button) と呼ばれます。

13.3.2　制御変数

チェックボタンの状態は、「制御変数」(control variable) と呼ばれるオブジェクトによって保持されます。制御変数というのは、ウィジェットが扱うことのできる、1 個のオブジェクトを保持することのできるオブジェクトのことです。

制御変数は、次のようなクラスから生成することができます。

`IntVar`	1 個の整数を保持することのできる制御変数を生成する。
`StringVar`	1 個の文字列を保持することのできる制御変数を生成する。

制御変数は、次のようなメソッドを持っています。

`set`	引数を自身に設定する。
`get`	自身が保持しているオブジェクトを返す。

13.3.3　チェックボタンの生成

チェックボタンは、`Checkbutton` というウィジェットクラスから生成されます。

`Checkbutton` には、1 個目の引数として親のウィジェットを渡して、さらに次のようなキーワード引数を渡します。

`text`	チェックボタンの上に表示する文字列。
`variable`	チェックボタンの状態として整数を保持する制御変数。

チェックボタンの初期状態は、制御変数の初期値によって決定されます。制御変数の初期値が 0 ならばチェックボタンの初期状態はオフで、1 ならばオンです。そして、チェックボタンは、オフからオンに切り替わったときに制御変数に 1 を設定して、オンからオフに切り替わったときに制御変数に 0 を設定します。

次のプログラムは、「状態の出力」というボタンがクリックされると、そのときのチェックボタンの状態を出力します。

プログラムの例　`checkbutton.py`

```
from tkinter import *

def clicked():
    print('状態は' + str(var.get()) + 'です。')

root = Tk()
var = IntVar()
var.set(0)
check = Checkbutton(root, text='私はチェックボタンです。',
    variable=var, font='Times 20')
check.pack()
button = Button(root, text='状態の出力', command=clicked,
    font='Times 20')
button.pack()
root.mainloop()
```

13.4 ラジオボタン

13.4.1 ラジオボタンの基礎

「ラジオボタン」(radio button) と呼ばれるボタンは、チェックボタンと同じように、オンの状態（チェックが入っている状態）とオフの状態（チェックが入っていない状態という二通りの状態を持っていて、クリックすることによってそれらの状態を切り替えることができます。ラジオボタンとチェックボタンとの相違点は、前者はグループで使われるというところにあります。

一つのグループを構成している 2 個以上のラジオボタンは、それらのうちのどれかをクリックすると、それがオンの状態になって、それまでオンの状態だったラジオボタンは、それと同時にオフになります。ですから、ラジオボタンは、いくつかの選択肢の中のうちのひとつを人間に選択させたい、というときに使うことができます。

13.4.2 ラジオボタンの生成

ラジオボタンは、`Ratiobutton` というウィジェットクラスから生成されます。

`Ratiobutton` には、1 個目の引数として親のウィジェットを渡して、さらに次のようなキーワード引数を渡します。

`text`	ラジオボタンの上に表示する文字列。
`variable`	ラジオボタンの状態を保持する制御変数。このキーワード引数で同一の制御変数が設定されている複数のラジオボタンは、ひとつのグループとして動作する。
`value`	ラジオボタンを識別するオブジェクト（整数または文字列）。制御変数には、選択されているラジオボタンを識別するオブジェクトが設定される。

ラジオボタンの状態は、制御変数に設定されているオブジェクトによって決定されます。制御変数の値と、`value` で設定されたオブジェクトとが等しいならば、そのラジオボタンの状態はオンで、等しくないならばオフです。そして、ラジオボタンは、自分がクリックされたときに、自分を識別するオブジェクトを制御変数に設定します。

次のプログラムは、「選択結果の出力」というボタンがクリックされると、そのときにオンになっているラジオボタンを識別する文字列を出力します。

プログラムの例 `radiobutton.py`

```
from tkinter import *

def clicked():
    print('選択結果は' + str(var.get()) + 'です。')

root = Tk()
var = StringVar()
var.set('カツ丼')
yakisoba = Radiobutton(root, text='焼きそば', value='焼きそば',
    variable=var, font='Times 20')
yakisoba.pack()
yudoufu = Radiobutton(root, text='湯豆腐', value='湯豆腐',
    variable=var, font='Times 20')
yudoufu.pack()
katsudon = Radiobutton(root, text='カツ丼', value='カツ丼',
    variable=var, font='Times 20')
katsudon.pack()
button = Button(root, text='選択結果の出力', command=clicked,
    font='Times 20')
button.pack()
root.mainloop()
```

13.5 メニュー

13.5.1 メニューの基礎

クリックされたときに選択肢を表示して、それらの選択肢のひとつがクリックされると、何らかの動作をする、というウィジェットは、「メニュー」(menu) と呼ばれます。

メニューを構成している個々の選択肢は、「メニュー項目」(menu item) と呼ばれます。

13.5.2　メニューの生成

メニューは、Menu というウィジェットクラスから生成されます。

Menu には、引数として親のウィジェットを渡します。

13.5.3　メニュー項目の追加

メニューは、add_command というメソッドを持っています。このメソッドは、メニュー項目をメニューに追加します。

add_command には、次のようなキーワード引数を渡すことができます。

label	メニュー項目の上に表示する文字列。
command	メニュー項目がクリックされたときに呼び出される関数またはメソッド。

13.5.4　メニューバー

ウィンドウのタイトルバーの下に取り付けられたメニューは、「メニューバー」(menu bar) と呼ばれます。

ウィンドウが持っている config を呼び出して、menu というキーワード引数で、そのメソッドにメニューを渡すと、そのメニューは、そのウィンドウのメニューバーになります。

次のプログラムは、メニューバーが取り付けられたウィンドウを表示します。

プログラムの例　menubar.py

```
from tkinter import *

def yakisoba():
    print('焼きそばがクリックされました。')

def katsugon():
    print('カツ丼がクリックされました。')

root = Tk()
menubar = Menu(root)
menubar.add_command(label='焼きそば', command=yakisoba)
menubar.add_command(label='カツ丼', command=katsugon)
root.config(menu=menubar)
root.mainloop()
```

13.5.5　サブメニュー

メニューは、何重にでも入れ子にすることができます。メニューの中に含まれているメニューは、「サブメニュー」(submenu) と呼ばれます。

サブメニューも、Menu というウィジェットクラスから生成されるウィジェットです。サブメニューを生成する場合、Menu には、親になるメニューを引数として渡します。

サブメニューは、メニューバーから切り離して、独立したウィンドウにする、ということも可能です。デフォルトでは、サブメニューは切り離しが可能なように設定されています。もしも切り離しができないように設定したいときは、Menu に対して、tearoff というキーワード引数で 0 という整数を渡します。

メニューが持っている add_cascade というメソッドを呼び出して、サブメニューを引数として渡すと、そのサブメニューがメニューに追加されます。

add_cascade には、次のようなキーワード引数を渡すことができます。

label	サブメニューの上に表示する文字列。
menu	サブメニューとして追加するメニュー。

メニューが持っている add_separator というメソッドは、サブメニューに対して、メニュー項目とメニュー項目とのあいだの区切りを追加します。

次のプログラムは、サブメニューを持つメニューバーが取り付けられたウィンドウを表示します。

プログラムの例　submenu.py

```
from tkinter import *
```

```
def yakisoba():
    print('焼きそばがクリックされました。')

def katsugon():
    print('カツ丼がクリックされました。')

def milk():
    print('ミルクがクリックされました。')

def coffee():
    print('コーヒーがクリックされました。')

def mizu():
    print('水がクリックされました。')

root = Tk()
menubar = Menu(root)
tabemono = Menu(menubar, tearoff=0)
tabemono.add_command(label='焼きそば', command=yakisoba)
tabemono.add_command(label='カツ丼', command=katsugon)
menubar.add_cascade(label='食べ物', menu=tabemono)
nomimono = Menu(menubar, tearoff=0)
nomimono.add_command(label='ミルク', command=milk)
nomimono.add_command(label='コーヒー', command=coffee)
nomimono.add_separator()
nomimono.add_command(label='水', command=mizu)
menubar.add_cascade(label='飲み物', menu=nomimono)
root.config(menu=menubar)
root.mainloop()
```

13.5.6 チェックボタンのメニュー項目

メニューに追加することのできるメニュー項目としては、通常のもののほかに、チェックボタンの機能を持つものや、ラジオボタンの機能を持つものがあります。

メニューが持っている add_checkbutton というメソッドは、チェックボタンの機能を持つメニュー項目をメニューに追加します。

add_checkbutton には、次のようなキーワード引数を渡すことができます。

label	メニュー項目の上に表示する文字列。
variable	チェックボタンの状態を保持する制御変数。

次のプログラムが表示するウィンドウは、チェックボタンの機能を持つメニュー項目を含むメニューを持っています。

プログラムの例 checkmenu.py

```
from tkinter import *

def condition():
    print('状態は' + str(var.get()) + 'です。')

root = Tk()
var = IntVar()
var.set(0)
menubar = Menu(root)
checkmenu = Menu(menubar, tearoff=0)
checkmenu.add_checkbutton(label='チェックボタン', variable=var)
checkmenu.add_command(label='状態の出力', command=condition)
menubar.add_cascade(label='チェックボタンのあるメニュー',
    menu=checkmenu)
root.config(menu=menubar)
root.mainloop()
```

13.5.7　ラジオボタンのメニュー項目

メニューが持っている add_radiobutton というメソッドは、ラジオボタンの機能を持つメニュー項目をメニューに追加します。

add_radiobutton には、次のようなキーワード引数を渡すことができます。

label	メニュー項目の上に表示する文字列。
variable	ラジオボタンの状態を保持する制御変数。
value	ラジオボタンを識別するオブジェクト（整数または文字列）。

次のプログラムが表示するウィンドウは、ラジオボタンの機能を持つメニュー項目を含むメニューを持っています。

プログラムの例　radiomenu.py

```
from tkinter import *

def selected():
    print('選択結果は' + str(var.get()) + 'です。')

root = Tk()
var = StringVar()
var.set('カツ丼')
menubar = Menu(root)
radiomenu = Menu(menubar, tearoff=0)
radiomenu.add_radiobutton(label='焼きそば', value='焼きそば',
    variable=var)
radiomenu.add_radiobutton(label='湯豆腐', value='湯豆腐',
    variable=var)
radiomenu.add_radiobutton(label='カツ丼', value='カツ丼',
    variable=var)
radiomenu.add_separator()
radiomenu.add_command(label='選択結果の出力', command=selected)
menubar.add_cascade(label='ラジオボタンのあるメニュー',
    menu=radiomenu)
root.config(menu=menubar)
root.mainloop()
```

第 14 章　ジオメトリーマネージャー

14.1　pack

14.1.1　pack についての復習

すでに第 10.2.6 項で簡単に説明したように、ウィジェットが持っている、自身の配置を管理するメソッドは、「ジオメトリーマネージャー」(geometry manager) と呼ばれます。

tkinter には、pack、grid、place という 3 種類のジオメトリーマネージャーがあります。

pack は、親のウィジェットを必要最小限に小さくして、その中に自身を詰め込む、という動作をするジオメトリーマネージャーです。

14.1.2　pack が受け取る引数

pack は、次のようなキーワード引数を受け取ることができます。

side	ウィジェットを詰め込む方向。
anchor	配置領域の中での位置。
fill	引き伸ばしの指定。
padx	x 軸方向の余白の大きさ。
pady	y 軸方向の余白の大きさ。
ipadx	x 軸方向の詰めものの大きさ。
ipady	y 軸方向の詰めものの大きさ。

14.1.3 詰め込みの方向

pack を使ってウィジェットを思ったとおりの位置に配置するためには、そのウィジェットを親に対してどういう方向へ詰め込むのかということを指定する必要があります。それを指定したいときは、side というキーワード引数で、方向を指定する文字列を pack に渡します。

side で渡すことのできる文字列のそれぞれには、TOP、BOTTOM、LEFT、RIGHT という識別子が束縛されています。それぞれの識別子は、親に対してウィジェットを詰め込む方向を示しています。デフォルトは TOP です。

pack によってウィジェットがどのように配置されのるかというのは、詰め込む順番に依存して決まります。指定された方向へウィジェットが詰め込まれると、それとは逆の方向に、仮想的な残りの領域ができます。そして、それ以降に詰め込まれるウィジェットは、その残りの領域を使うことになります。ですから、ウィジェットを詰め込む方向というのは、それ以降に詰め込むウィジェットの位置を決定するためのものだと考えるとわかりやすいでしょう。

次のプログラムは、詰め込みの方向を指定して三つのプッシュボタンをウィンドウに詰め込みます。

プログラムの例 side.py

```
from tkinter import *
root = Tk()
button = Button(root, text='1:LEFT', font='Times 20')
button.pack(side=LEFT)
button = Button(root, text='2:TOP', font='Times 20')
button.pack(side=TOP)
button = Button(root, text='3:LEFT', font='Times 20')
button.pack(side=LEFT)
root.mainloop()
```

ひとつ目のプッシュボタンは左の方向へ詰め込まれていますので、それ以降に詰め込まれるウィジェットは、その右側の領域を使うことになります。そして、二つ目のプッシュボタンは上の方向へ詰め込まれていますので、それ以降に詰め込まれるウィジェットは、その下の領域を使うことになります。

14.1.4 配置領域の中での位置

親の上に残されている、ウィジェットを詰め込むことのできる領域のことを、「配置領域」と呼ぶことにしましょう。そして、詰め込まれるウィジェットそのものの領域のことを「表示領域」と呼ぶことにしましょう。

表示領域が配置領域よりも小さい場合、ウィジェットは、デフォルトでは配置領域の中央に配置されます。しかし、配置領域の中のそれ以外の位置にウィジェットを配置することも可能です。それをしたいときは、anchor というキーワード引数で、ウィジェットを配置する位置を指定する文字列を pack に渡します。

anchor で渡すことのできる文字列には、次のような識別子が束縛されています。

```
NW     N     NE
W   CENTER    E
SW     S     SE
```

ちなみに、CENTER を除くそれぞれの識別子は、east、west、north、south、north east、south east、north west、south west という、方位を示す言葉の頭字語です。

次のプログラムは、「左右に長いプッシュボタン」というプッシュボタンの下に、左右の長さがそれよりも短い二つのプッシュボタンを配置します。「デフォルト」というプッシュボタンは anchor がデフォルトで、「右寄せ」というプッシュボタンは、E という識別子が束縛されている文字列を anchor で渡しています。

プログラムの例 anchor.py

```
from tkinter import *
root = Tk()
button = Button(root, text='左右に長いプッシュボタン',
    font='Times 20')
button.pack()
button = Button(root, text='デフォルト', font='Times 20')
```

```
button.pack()
button = Button(root, text=' 右寄せ', font='Times 20')
button.pack(anchor=E)
root.mainloop()
```

14.1.5　引き延ばし

ウィジェットの大きさというのは、基本的には、その内部に表示されるものの大きさによって決まります。しかし、いくつかのウィジェットを並べて配置する場合には、それらの大きさをそろえないと、見栄えがよくありません。

並んでいるいくつかのウィジェットの大きさをそろえたいときは、小さなウィジェットを大きく引き延ばす、という技法を使います。ウィジェットの大きさを引き延ばしたいときは、 fill というキーワード引数で、引き伸ばす方向を指定する文字列を pack に渡します。

fill で渡すことのできる文字列には、 NONE、 X、 Y、 BOTH という識別子が束縛されています。 NONE は引き延ばさない、 X は x 軸方向にのみ引き延ばす、 Y は y 軸方向にのみ引き延ばす、 BOTH はどちらの方向にも引き延ばすという意味で、デフォルトは NONE です。

次のプログラムは、「左右に長いプッシュボタン」という文字列を表示するプッシュボタンの下に、表示する文字列がそれよりも短い二つのプッシュボタンを配置します。「デフォルト」というプッシュボタンは fill がデフォルトで、「引き伸ばし」というプッシュボタンは、 X という識別子が束縛されている文字列を fill で渡しています。

プログラムの例　fill.py

```
from tkinter import *
root = Tk()
button = Button(root, text=' 左右に長いプッシュボタン',
    font='Times 20')
button.pack()
button = Button(root, text=' デフォルト', font='Times 20')
button.pack()
button = Button(root, text=' 引き伸ばし', font='Times 20')
button.pack(fill=X)
root.mainloop()
```

14.1.6　余白

ウィジェットの周囲に余白を作りたいときは、 padx または pady というキーワード引数で、余白の大きさを pack に渡します。 padx を渡すことによって x 軸方向の余白の大きさを、 pady を渡すことによって y 軸方向の余白の大きさを指定することができます。余白の大きさの単位はピクセルです。

次のプログラムは、二つのプッシュボタンをウィンドウに詰め込みます。上のプッシュボタンは余白がデフォルトで、下のプッシュボタンには上下左右に 50 ピクセルの余白が指定されています。

プログラムの例　pad.py

```
from tkinter import *
root = Tk()
button = Button(root, text=' デフォルト', font='Times 20')
button.pack()
button = Button(root, text=' 上下左右に余白', font='Times 20')
button.pack(padx=50, pady=50)
root.mainloop()
```

14.1.7　詰めもの

ウィジェットの上に表示されるテキストなどの周囲に詰めものを詰めることによってウィジェットの大きさを大きくする、ということも可能です。それをしたいときは、 ipadx または ipady というキーワード引数で、詰め物の大きさを pack に渡します。 ipadx を渡すことによって x 軸方向の詰めものの大きさを、 ipady を渡すことによって y 軸方向の詰めものの大きさを指定することができます。詰めものの大きさの単位はピクセルです。

次のプログラムは、二つのプッシュボタンをウィンドウに詰め込みます。上のプッシュボタンは詰めものがデフォルトで、下のプッシュボタンには上下左右に 50 ピクセルの詰めものが指定

されています。

プログラムの例 ipad.py

```
from tkinter import *
root = Tk()
button = Button(root, text='デフォルト', font='Times 20')
button.pack()
button = Button(root, text='上下左右に詰めもの',
    font='Times 20')
button.pack(ipadx=50, ipady=50)
root.mainloop()
```

14.2 grid

14.2.1 grid の基礎

grid は、ウィジェットを 2 次元の表の形に並べるジオメトリーマネージャーです。

2 次元の表を構成するそれぞれの部分は、「セル」(cell) と呼ばれます。そして、水平方向にセルを並べたものは「行」(row) と呼ばれ、垂直方向にセルを並べたものは「列」(column) と呼ばれます。

14.2.2 grid が受け取る引数

grid は、次のようなキーワード引数を受け取ることができます。

row	行の番号（もっとも上の行は 0）。
column	列の番号（もっとも左の列は 0）。
rowspan	何行にまたがって配置するか。
columnspan	何列にまたがって配置するか。
sticky	配置領域の中にどのように配置するか。
padx	x 軸方向の余白の大きさ。
pady	y 軸方向の余白の大きさ。
ipadx	x 軸方向の詰めものの大きさ。
ipady	y 軸方向の詰めものの大きさ。

14.2.3 行と列の番号

grid を使ってウィジェットを配置するためには、最低限、何行目の何列目にそれを配置するのかという位置を指定する必要があります。

位置の指定には、row と column というキーワード引数を使います。row は行の番号で、もっとも上の行を 0 と数えます。column は行の番号で、もっとも左の列を 0 と数えます。

次のプログラムは、grid を使って、4 行 3 列の表の形にプッシュボタンを配置します。

プログラムの例 grid.py

```
from tkinter import *
root = Tk()
for i in range(4):
    for j in range(3):
        button = Button(root,
            text=str(i)+'行目の'+str(j)+'列目', font='Times 20')
        button.grid(row=i, column=j)
root.mainloop()
```

14.2.4 複数のセルにまたがった配置

pack は、垂直方向または水平方向に並んでいる連続した複数のセルにひとつのウィジェットを配置するということもできます。

rowspan というキーワード引数でプラスの整数 n を渡すと、ウィジェットは、row で指定された位置から下向きに n 個の連続したセルに配置されます。

同じように、columnspan というキーワード引数でプラスの整数 n を渡すと、ウィジェットは、column で指定された位置から右向きに n 個の連続したセルに配置されます。

次のプログラムは、4 個のプッシュボタンをウィンドウの上に配置します。そのうちのひとつは垂直方向の二つのセルに配置され、別のひとつは水平方向の二つのセルに配置されます。

プログラムの例　span.py

```
from tkinter import *
root = Tk()
button = Button(root, text='0 行目と 1 行目の 0 列目',
    font='Times 20')
button.grid(row=0, column=0, rowspan=2)
button = Button(root, text='0 行目の 1 列目', font='Times 20')
button.grid(row=0, column=1)
button = Button(root, text='0 行目の 2 列目', font='Times 20')
button.grid(row=0, column=2)
button = Button(root, text='1 行目の 1 列目と 2 列目',
    font='Times 20')
button.grid(row=1, column=1, columnspan=2)
root.mainloop()
```

14.2.5　配置領域の中での表示方法

pack の場合、配置領域の中での位置を指定するキーワード引数は anchor で、ウィジェットを引き伸ばす方向を指定するキーワード引数は fill でした。それに対して grid の場合は、配置領域の中での位置も、ウィジェットを引き伸ばす方向も、 sticky というひとつのキーワード引数を使って指定します。

配置領域の中での位置を指定する方法は、pack の anchor と同じです。N 、NE、E などの方位を示す識別子が束縛されている文字列を、 sticky というキーワード引数で grid に渡すと、ウィジェットは、方位で指定された位置に表示されます。

ウィジェットを引き伸ばす場合は、識別子が束縛されている文字列を次のように連結します。

N+S	垂直方向に引き伸ばす。
E+W	水平方向に引き伸ばす。
N+E+S+W	垂直方向と水平方向の両方に引き伸ばす。

次のプログラムは、「左右に長いプッシュボタン」というプッシュボタンの下に、左右の長さがそれよりも短い三つのプッシュボタンを配置します。「デフォルト」というプッシュボタンは sticky がデフォルトで、「右寄せ」というプッシュボタンは、 E という識別子が束縛されている文字列を sticky で渡していて、「引き伸ばし」というプッシュボタンは、 E+W という二つの識別子のそれぞれに束縛されている文字列を連結したものを sticky で渡しています。

プログラムの例　sticky.py

```
from tkinter import *
root = Tk()
button = Button(root, text=' 左右に長いプッシュボタン',
    font='Times 20')
button.grid(row=0, column=0)
button = Button(root, text=' デフォルト', font='Times 20')
button.grid(row=1, column=0)
button = Button(root, text=' 右寄せ', font='Times 20')
button.grid(row=2, column=0, sticky=E)
button = Button(root, text=' 引き伸ばし', font='Times 20')
button.grid(row=3, column=0, sticky=W+E)
root.mainloop()
```

14.3　place

14.3.1　place の基礎

place は、指定された座標によってウィジェットを配置するジオメトリーマネージャーです。

place を使ってウィジェットを配置する場合には、配置するウィジェットの親となるウィジェットの大きさを設定しておく必要があります。

14.3.2 ルートウィジェットの大きさ

ウィジェットの大きさは、ウィジェットクラスに対して、width というキーワード引数で横の長さ、height というキーワード引数で縦の長さを渡すことによって指定することができるのですが、ルートウィジェットの大きさを指定する場合は、それらの引数を、ウィジェットクラスではなくて、生成されたルートウィジェットが持っている config に渡す必要があります。

次のプログラムは、横の長さが 600 ピクセル、縦の長さが 400 ピクセルのルートウィジェットを表示します。

プログラムの例 rootsize.py

```
from tkinter import *
root = Tk()
root.config(width=600, height=400)
root.mainloop()
```

14.3.3 絶対的な座標によるウィジェットの配置

親のウィジェットの上の位置を絶対的な座標で指定してウィジェットを配置したいときは、x と y というキーワード引数で、絶対的な x 座標と y 座標を place に渡します。

次のプログラムは、絶対的な座標を指定して二つのプッシュボタンを配置します。

プログラムの例 xy.py

```
from tkinter import *
root = Tk()
root.config(width=600, height=400)
button = Button(root, text='(200, 100)', font='Times 20')
button.place(x=200, y=100)
button = Button(root, text='(300, 200)', font='Times 20')
button.place(x=300, y=200)
root.mainloop()
```

14.3.4 相対的な座標によるウィジェットの配置

親のウィジェットの上の位置を相対的な座標で指定してウィジェットを配置したいしたいときは、relx と rely というキーワード引数で、相対的な x 座標と y 座標を place に渡します。相対的な座標は、0 から 1 までの浮動小数点数です。

次のプログラムは、相対的な座標を指定して二つのプッシュボタンを配置します。

プログラムの例 relxrely.py

```
from tkinter import *
root = Tk()
root.config(width=600, height=400)
button = Button(root, text='(0.333, 0.25)', font='Times 20')
button.place(relx=0.333, rely=0.25)
button = Button(root, text='(0.5, 0.5)', font='Times 20')
button.place(relx=0.5, rely=0.5)
root.mainloop()
```

14.4 フレーム

14.4.1 フレームの基礎

tkinter のウィジェットは、pack、grid、place という 3 種類のジオメトリーマネージャーを持っているわけですが、それらのうちで万能のものというのは存在しません。ですから、どのジオメトリーマネージャーを選んでも望みどおりにウィジェットを配置することができない、ということもしばしばあります。しかし、そのような問題の多くは、「フレーム」(frame) と呼ばれるウィジェットを使うことによって解決することができます。

フレームというのは、その上にウィジェットを配置するということだけを機能として持つウィジェットのことです。

ウィンドウの上に複数のフレームを配置して、それらのフレームの上に複数のウィジェットを配置することによって、どんなに複雑な配置でも可能になります。

14.4.2　フレームの生成

フレームは、Frame というウィジェットクラスから生成されます。

Frame には、1 個目の引数として親のウィジェットを渡して、さらに次のようなキーワード引数を渡します。

width	横の長さ。
height	縦の長さ。
borderwidth	境界線の幅。デフォルトは 0。
relief	境界線のスタイル。
padx	x 軸方向の詰めものの大きさ。
pady	y 軸方向の詰めものの大きさ。

次のプログラムは、二つのフレームを左右に並べて、左のフレームの上には 4 行 3 列でプッシュボタンを並べて、右のフレームの上には 3 行 4 列でプッシュボタンを並べています。

プログラムの例　frame.py

```
from tkinter import *
root = Tk()
left = Frame(root, padx=50, pady=50)
left.pack(side=LEFT)
right = Frame(root, padx=50, pady=50)
right.pack()
for i in range(4):
    for j in range(3):
        button = Button(left, text=str(i)+', '+str(j),
            font='Times 20')
        button.grid(row=i, column=j)
for i in range(3):
    for j in range(4):
        button = Button(right, text=str(i)+', '+str(j),
            font='Times 20')
        button.grid(row=i, column=j)
root.mainloop()
```

14.4.3　境界線のスタイル

borderwidth というキーワード引数で Frame に 1 以上の数値を渡すと、その数値を幅とする境界線が表示されます。

境界線のスタイルは、relief というキーワード引数で、次の識別子を Frame に渡すことによって指定することができます。

```
FLAT   RAISED   SUNKEN   GROOVE   RIDGE
```

次のプログラムは、境界線のスタイルが異なる 5 個のフレームを表示します。

プログラムの例　relief.py

```
from tkinter import *
root = Tk()
for relief in (FLAT, RAISED, SUNKEN, GROOVE, RIDGE):
    frame = Frame(root, borderwidth=6, relief=relief)
    frame.pack(side=LEFT, padx=20, pady=20)
    label = Label(frame, text=relief, font='Times 20')
    label.pack()
root.mainloop()
```

14.4.4　ラベルフレーム

フレームを生成するウィジェットクラスとしては、Frame のほかに、LabelFrame というものもあります。

LabelFrame が生成するフレームは、「ラベルフレーム」(label frame) と呼ばれます。ラベルフレームは、その名前のとおり、ラベルを表示することのできるフレームです。

LabelFrame には、1 個目の引数として親のウィジェットを渡して、さらに、ラベルにする文

字列とラベルの位置を次のキーワード引数で渡します。

text	ラベルにする文字列。
labelanchor	ラベルの位置。nw、n、ne、en、e、es というような、方角を示す文字列で指定する。

ちなみに、Frame に渡すことのできるキーワード引数は、LabelFrame にも渡すことができます。

次のプログラムは、ラベルの位置が異なる 12 個のラベルフレームを表示します。

プログラムの例　labelframe.py

```
from tkinter import *
root = Tk()
for anchor in ('nw', 'n', 'ne', 'en', 'e', 'es', 'se',
        's', 'sw', 'ws', 'w', 'wn'):
    frame = LabelFrame(root, text=anchor, labelanchor=anchor)
    frame.pack(side=LEFT, padx=5, pady=5, ipadx=25, ipady=25)
root.mainloop()
```

第 15 章　文字列入力ウィジェット

15.1　文字列入力ウィジェットの基礎

15.1.1　エントリーとテキストウィジェット

この章では、人間がそれを使って文字列を入力したり編集したりすることのできるウィジェットを紹介したいと思います。

tkinter には、文字列の入力や編集に使うことのできるウィジェットとして、次の二つのものがあります。

- エントリー (entry)
- テキストウィジェット (text widget)

15.1.2　フォーカス

キーボードというのはひとつのコンピュータにひとつだけしかありませんので、表示されているいくつかのウィジェットのうちで、キーボードから入力された文字を受け取ることができるものは、ひとつの時点ではひとつだけです。キーボードから入力された文字を受け取ることができる状態にあるウィジェットは、「フォーカス」(focus) が設定されている、と言われます。

フォーカスの設定は、マウスやキーボードの操作によって切り替えることも可能ですが、プログラムの側で特定のウィジェットにフォーカスを設定することも可能です。

ウィジェットは、focus_set というメソッドを持っています。特定のウィジェットにフォーカスを設定したいときは、そのウィジェットが持っているこのメソッドを呼び出します。

15.2　エントリー

15.2.1　エントリーの基礎

1 行だけの文字列（つまり改行を含まない文字列）を人間から受け取るウィジェットは、「エントリー」(entry) と呼ばれます。

15.2.2　エントリーの生成

エントリーは、Entry というウィジェットクラスから生成されます。

Entry には、1 個目の引数として親のウィジェットを渡します。

デフォルトでは、エントリーの横の長さは、20 文字を表示することのできる長さに設定されています。それとは異なる長さを設定したいときは、width というキーワード引数で、表示することのできる文字数を Entry に渡します。なお、入力することのできる文字列の長さは、エントリーの横の長さには制限されません。

15.2.3　エントリーからの文字列の取り出し

エントリーに入力されている文字列を取り出したいときは、get というメソッドを使います。get は、エントリーに入力されている文字列を戻り値として返します。

次のプログラムは、「転送」というプッシュボタンが押されると、そのときにエントリーに入力されている文字列をラベルへ転送します。

プログラムの例　entry.py

```
from tkinter import *

def transfer():
    label.config(text=entry.get())

root = Tk()
entry = Entry(root, width=30, font='Times 20')
entry.pack()
entry.focus_set()
button = Button(root, text=' 転送', command=transfer,
    font='Times 20')
button.pack()
label = Label(root, font='Times 20')
label.pack()
root.mainloop()
```

15.2.4　エンターキーのバインディング

「エンターキーが押された」というイベントにイベントハンドラーをバインドしておくことによって、エンターキーが押されたときにエントリーに入力された文字列を処理する、ということもできます。

エンターキーのキーシムは Return ですので、「エンターキーが押された」というイベントをバインドするためのイベントパターンは、

```
<Key-Return>
```

と書くことができます。

次のプログラムは、エンターキーが押されると、そのときにエントリーに入力されている文字列をラベルへ転送します。

プログラムの例　enter.py

```
from tkinter import *

def transfer(event):
    label.config(text=entry.get())

root = Tk()
entry = Entry(root, width=30, font='Times 20')
entry.bind('<Key-Return>', transfer)
entry.pack()
entry.focus_set()
label = Label(root, font='Times 20')
label.pack()
root.mainloop()
```

15.2.5　エントリーへの制御変数の設定

Entry に対しては、textvariable というキーワード引数で、文字列を保持することのできる制御変数を渡すことができます。そうすることによって制御変数をエントリーに設定しておくと、そのエントリーに入力された文字列は、制御変数の中に自動的に設定されます。

Label に対しても、Entry と同様に、textvariable というキーワード引数で、文字列を保持することのできる制御変数を渡すことができます。制御変数が設定されたラベルは、その制御変数に設定されている文字列を表示することになります。

次のプログラムは、エントリーに入力された文字列を、入力と同時にラベルに表示します。

プログラムの例　textvariable.py

```
from tkinter import *
```

```
root = Tk()
var = StringVar()
entry = Entry(root, width=30, textvariable=var, font='Times 20')
entry.pack()
entry.focus_set()
label = Label(root, textvariable=var, font='Times 20')
label.pack()
root.mainloop()
```

15.3 テキストウィジェット

15.3.1 テキストウィジェットの基礎

改行を含む文字列、つまりいくつかの行から構成される文字列を人間から受け取るウィジェットは、「テキストウィジェット」(text widget) と呼ばれます。

15.3.2 テキストウィジェットの生成

テキストウィジェットは、Text というウィジェットクラスから生成されます。

Text には、1 個目の引数として親のウィジェットを渡します。さらに Text は、次のようなキーワード引数を受け取ります。

width　横の長さ。1 行のうちで表示することのできる文字数。

height　縦の長さ。表示することのできる行数。

wrap　width で設定した文字数よりも長い行を折り返すかどうかということを指定する文字列。次の識別子を使うことができる。

- CHAR　文字単位で折り返す。
- WORD　単語単位で折り返す。
- NONE　折り返さない（左右にスクロールして表示）。

次のプログラムは、テキストウィジェットを持つウィンドウを表示します。

プログラムの例　textwidget.py

```
from tkinter import *
root = Tk()
text = Text(root, width=30, height=10, wrap=WORD,
    font='Times 20')
text.pack()
text.focus_set()
root.mainloop()
```

15.3.3 テキストウィジェットからの文字列の取り出し

テキストウィジェットに入力されている文字列を取り出したいときは、get というメソッドを使います。get は、テキストウィジェットに入力されている文字列を戻り値として返します。

エントリーの get とは違って、テキストウィジェットの get には 2 個の引数を渡す必要があります。それらの引数は、取り出す文字列の範囲を指定するものです。1 個目の引数は開始位置を示す文字列で、2 個目の引数は終了位置を示す文字列です。

文字列の中の位置を示す文字列は、行の番号と文字の番号をドット (.) で区切って並べたものです。先頭の行は 1 行目と数え、左端の文字は 0 文字目と数えます。たとえば、5.8 という文字列は、5 行目の 8 文字目という意味になります。文字列の末尾を指定したいときは、END という識別子が束縛されている文字列を指定します。

次のプログラムは、「転送」というプッシュボタンが押されると、そのときにテキストウィジェットに入力されている文字列の全体をラベルへ転送します。

プログラムの例　get.py

```
from tkinter import *

def transfer():
    label.config(text=text.get("1.0", END))
```

```
root = Tk()
text = Text(root, width=30, height=5, wrap=CHAR,
    font='Times 20')
text.pack()
text.focus_set()
button = Button(root, text=' 転送', command=transfer,
    font='Times 20')
button.pack()
label = Label(root, font='Times 20')
label.pack()
root.mainloop()
```

15.3.4　スクロールバーの生成

ウィジェットをスクロールさせるために使われるウィジェットは、「スクロールバー」(scroll bar) と呼ばれます。

スクロールバーは、Scrollbar というウィジェットクラスから生成されます。

Scrollbar には、1 個目の引数として親のウィジェットを渡して、さらに、orient というキーワード引数で、スクロールの方向を指定する文字列を渡します。

次の識別子は、スクロールの方向を指定する文字列に束縛されています。

HORIZONTAL　水平方向。

VERTICAL　垂直方向。

15.3.5　ウィジェットとスクロールバーの連動

ウィジェットとスクロールバーとを連動させるためには、config を呼び出すことによって、それぞれが持っているメソッドを相互に設定する必要があります。

ウィジェットとスクロールバーとを連動させるためには、そのスクロールバーが持っている config を呼び出して、command というキーワード引数で、そのウィジェットが持っている次のメソッドを渡す必要があります。

xview　水平方向のスクロールバーに渡すメソッド。

yview　垂直方向のスクロールバーに渡すメソッド。

ウィジェットとスクロールバーとを連動させるためには、さらに、そのウィジェットが持っている config を呼び出して、次のキーワード引数で、そのスクロールバーが持っている set というメソッドを渡す必要があります。

xscrollcommand　水平方向のスクロールバーの set を渡すキーワード引数。

yscrollcommand　垂直方向のスクロールバーの set を渡すキーワード引数。

次のプログラムは、スクロールバーを使ってスクロールさせることのできるテキストウィジェットを持つウィンドウを表示します。

プログラムの例　scroll.py

```
from tkinter import *
root = Tk()
text = Text(root, width=30, height=5, wrap=NONE,
    font='Times 20')
text.grid(row=0, column=0)
horizontal = Scrollbar(root, orient=HORIZONTAL)
horizontal.config(command=text.xview)
horizontal.grid(row=1, column=0, sticky=E+W)
vertical = Scrollbar(root, orient=VERTICAL)
vertical.config(command=text.yview)
vertical.grid(row=0, column=1, sticky=N+S)
text.config(xscrollcommand=horizontal.set)
text.config(yscrollcommand=vertical.set)
text.focus_set()
root.mainloop()
```

第 16 章　定型的なダイアログボックス

16.1 定型的なダイアログボックスの基礎

16.1.1 ダイアログボックスとは何か

人間との対話のためにプログラムが一時的に画面に表示するウィンドウは、「ダイアログボックス」(dialog box) と呼ばれます。

ダイアログボックスというのはウィンドウの一種ですので、ダイアログボックスも、ウィンドウを生成するウィジェットクラスを呼び出すことによって生成することができます。そのような方法によるダイアログボックスの作り方については、第 17.3 節で説明することにしたいと思います。

しかし、もっと簡単にダイアログボックスを表示する方法もあります。頻繁に使われる定型的なダイアログボックスについては、ただ単にメソッドを呼び出すだけで表示することができます。

16.1.2 定型的なダイアログボックスの分類

メソッドを呼び出すだけで表示することのできる定型的なダイアログボックスは、次の 3 種類に分類することができます。

- メッセージボックス (message box)
- ファイル選択ダイアログボックス (file dialog)
- 色選択ダイアログボックス (color chooser)

16.2 メッセージボックス

16.2.1 メッセージボックスの基礎

人間に対してメッセージを伝えたり、人間に対して単純な問い合わせをしたりするときに使われる定型的なダイアログボックスは、「メッセージボックス」(message box) と呼ばれます。

メッセージボックスを表示するメソッドは、

```
tkinter.messagebox
```

というライブラリーの中で定義されていますので、プログラムの先頭に、

```
from tkinter.messagebox import *
```

と書いておけば、それらのメソッドを呼び出すことができるようになります。

16.2.2 メッセージボックスを表示するメソッド

メッセージボックスを表示するメソッドには、次のようなものがあります。

`showinfo`	情報を表示するメッセージボックス。
`showwarning`	警告を表示するメッセージボックス。
`showerror`	エラーメッセージを表示するメッセージボックス。
`askyesno`	「はい」または「いいえ」で答える質問を表示するメッセージボックス。戻り値は真偽値で、「はい」ならば真、「いいえ」ならば偽。
`askquestion`	「はい」または「いいえ」で答える質問を表示するメッセージボックス。戻り値は文字列で、「はい」ならば `yes`、「いいえ」ならば `no`。
`askokcancel`	「`OK`」または「キャンセル」で答える質問を表示するメッセージボックス。戻り値は真偽値で、「`OK`」ならば真、「キャンセル」ならば偽。
`askretrycancel`	「再試行」または「キャンセル」で答える質問を表示するメッセージボックス。戻り値は真偽値で、「再試行」ならば真、「キャンセル」ならば偽。

次のプログラムは、プッシュボタンがクリックされると、「はい」または「いいえ」で答える質問を表示します。そして、「はい」と答えたか「いいえ」と答えたかということに応じて、異なる返答を表示します。

プログラムの例 `askyesno.py`

```
from tkinter import *
from tkinter.messagebox import *

def askandanswer():
```

```
    if askyesno('質問', 'あなたはプログラミングが好きですか？'):
        showinfo('回答', 'それはすばらしい。')
    else:
        showinfo('回答', 'それは残念です。')

root = Tk()
button = Button(root, text='質問と回答', command=askandanswer,
    font='Times 20')
button.pack()
root.mainloop()
```

16.3　ファイル選択ダイアログボックス

16.3.1　ファイル選択ダイアログボックスの基礎

ファイルを人間に選択してもらうときに使われる定型的なダイアログボックスは、「ファイル選択ダイアログボックス」(message box) と呼ばれます。

ファイル選択ダイアログボックスを表示するメソッドは、

```
tkinter.filedialog
```

というライブラリーの中で定義されていますので、プログラムの先頭に、

```
from tkinter.filedialog import *
```

と書いておけば、それらのメソッドを呼び出すことができるようになります。

16.3.2　ファイル選択ダイアログボックスを表示するメソッド

ファイル選択ダイアログボックスを表示するメソッドには、次の二つがあります。

`askopenfilename`	すでに存在するファイルを選択するためのファイル選択ダイアログボックスを表示する。存在しないファイルが選択された場合は、メッセージボックスでそれを知らせる。
`asksaveasfilename`	データを保存する対象となるファイルを選択するためのファイル選択ダイアログボックスを表示する。すでに存在するファイルが選択された場合は、それを上書きしてもいいかどうかをメッセージボックスで尋ねる。

これらのメソッドは、ダイアログボックスが「開く」または「保存」のプッシュボタンで閉じられた場合は、選択されたファイルの絶対パス名を戻り値として返します。「キャンセル」のプッシュボタンで閉じられた場合は、空文字列を返します。

次のプログラムは、すでに存在するファイルを選択するためのファイル選択ダイアログボックスを表示して、選択されたファイルの絶対パス名を表示します。

プログラムの例　askopenfilename.py

```
from tkinter import *
from tkinter.messagebox import *
from tkinter.filedialog import *

def selectexistingfile():
    filename = askopenfilename()
    showinfo('選択結果', filename)

root = Tk()
button = Button(root, text='既存ファイル選択',
    command=selectexistingfile, font='Times 20')
button.pack()
root.mainloop()
```

次のプログラムは、データを保存する対象となるファイルを選択するためのファイル選択ダイアログボックスを表示して、選択されたファイルの絶対パス名を表示します。

プログラムの例　asksaveasfilename.py

```
from tkinter import *
from tkinter.messagebox import *
```

```
from tkinter.filedialog import *

def selectsaveasfile():
    filename = asksaveasfilename()
    showinfo(' 選択結果', filename)

root = Tk()
button = Button(root, text=' 保存用ファイル選択',
    command=selectsaveasfile, font='Times 20')
button.pack()
root.mainloop()
```

16.3.3 ファイル選択ダイアログボックスのキーワード引数

ファイル選択ダイアログボックスを表示するメソッドは、次のようなキーワード引数を受け取ります。

filetypes	選択の対象となるファイルのタイプを選択するために使われる、 [(ラベル, パターン), ···] という形式のリスト。「ラベル」は、タイプの説明として表示される文字列、「パターン」は、拡張子を示す、 .py のような文字列。
initialdir	最初に表示されるディレクトリ（フォルダー）。
initialfile	ファイル名のエントリーにデフォルトで入力されるファイル名。
defaultextension	入力されたファイル名にドット (.) が含まれていない場合に自動的に付加される拡張子。
title	タイトルバーに表示される文字列。
parent	親にするウィンドウ。ダイアログボックスは親のウィンドウの上に表示される。デフォルトはルートウィジェット。

次のプログラムは、画像ファイル（JPEG、PNG、GIF）を選択するためのファイル選択ダイアログボックスを表示します。

プログラムの例 imagefile.py

```
from tkinter import *
from tkinter.messagebox import *
from tkinter.filedialog import *

def selectimagefile():
    filename = askopenfilename(
        title=' 画像ファイルを開く',
        initialdir='c:\\',
        filetypes=[('JPEG', '.jpg'),
                   ('PNG', '.png'),
                   ('GIF', '.gif'),
                   ('All Files', '.*')])
    showinfo(' 選択結果', filename)

root = Tk()
button = Button(root, text=' 画像ファイル選択',
    command=selectimagefile, font='Times 20')
button.pack()
root.mainloop()
```

16.4 色選択ダイアログボックス

16.4.1 色選択ダイアログボックスの基礎

色を人間に選択してもらうときに使われる定型的なダイアログボックスは、「色選択ダイアログボックス」(color chooser) と呼ばれます。

色選択ダイアログボックスを表示するメソッドは、

```
tkinter.filechooser
```

というライブラリーの中で定義されていますので、プログラムの先頭に、

```
from tkinter.filechooser import *
```

と書いておけば、そのメソッドを呼び出すことができるようになります。

16.4.2　色選択ダイアログボックスを表示するメソッド

色選択ダイアログボックスは、askcolor というメソッドを呼び出すことによって表示することができます。

askcolor は、次のようなキーワード引数を受け取ります。

color	最初に選択されている色。デフォルトは明るい灰色。
title	タイトルバーに表示される文字列。
parent	親にするウィンドウ。ダイアログボックスは親のウィンドウの上に表示される。デフォルトはルートウィジェット。

次のプログラムは、色選択ダイアログボックスを使うことによって、ラベルの背景色を変更することができます。

プログラムの例　colorchooser.py

```
from tkinter import *
from tkinter.messagebox import *
from tkinter.colorchooser import *

def selectcolor():
    selected = askcolor(title='背景色の変更', color=color)
    if selected[1] != None:
        label.config(text=selected[1], background=selected[1])

color='#99ff99'
root = Tk()
label = Label(root, text=color, background=color,
    font='Times 20')
label.pack(ipadx=50, ipady=50)
button = Button(root, text='背景色の変更', command=selectcolor,
    font='Times 20')
button.pack()
root.mainloop()
```

第 17 章　トップレベルウィンドウ

17.1　トップレベルウィンドウの基礎

17.1.1　トップレベルウィンドウの基礎

第 10.2.2 項で説明したように、tkinter では、プログラムが表示するウィンドウのうちで、もっとも中心となるものを、「ルートウィジェット」(root widget) と呼びます。

それに対して、ルートウィジェット以外のウィンドウは、「トップレベルウィンドウ」(top-level window) と呼ばれます。

17.1.2　トップレベルウィンドウの生成

トップレベルウィンドウは、Toplevel というウィジェットクラスから生成されます。

通常、ウィジェットクラスには、引数として親のウィジェットを渡す必要があるわけですが、ルートウィジェットと同じように、トップレベルウィンドウも、その必要はありません。

次のプログラムは、プッシュボタンが押されるたびに、トップレベルウィンドウを生成します。

プログラムの例　toplevel.py

```
from tkinter import *

def create_toplevel():
    Toplevel()
```

```
root = Tk()
button = Button(root, text='トップレベルウィンドウの生成',
    command=create_toplevel, font='Times 20')
button.pack()
root.mainloop()
```

17.2 トップレベルウィンドウへのウィジェットの取り付け

17.2.1 トップレベルウィンドウへの通常のウィジェットの取り付け

ルートウィジェットと同じように、トップレベルウィンドウにも、さまざまなウィジェットを取り付けることができます。

ウィジェットをトップレベルウィンドウに取り付ける方法は、ウィジェットをルートウィジェットに取り付ける方法と同じです。つまり、ウィジェットクラスを呼び出して、トップレベルウィンドウを親のウィジェットとして渡して、ジオメトリーマネージャーを呼び出せばいいのです。

次のプログラムは、プッシュボタンが押されるたびに、プッシュボタンが取り付けられたトップレベルウィンドウを生成します。

プログラムの例 buttontoplevel.py

```
from tkinter import *

class ButtonToplevel:
    def __init__(self):
        global count
        count += 1
        self.id = count
        toplevel = Toplevel()
        button = Button(toplevel, text='私について',
            command=self.aboutme, font='Times 20')
        button.pack()
    def aboutme(self):
        print('私は' + str(self.id) +
            '番目のトップレベルウィンドウです。')

count = 0
root = Tk()
button = Button(root, text='トップレベルウィンドウの生成',
    command=ButtonToplevel, font='Times 20')
button.pack()
root.mainloop()
```

17.2.2 トップレベルウィンドウへのメニューバーの取り付け

メニューバーも、ルートウィジェットだけではなくて、トップレベルウィンドウにも取り付けることができます。

トップレベルウィンドウにメニューバーを取り付ける方法は、第 13.5.4 項で説明した、ルートウィジェットにメニューバーを取り付ける方法と同じです。つまり、トップレベルウィンドウを親とするメニューを作って、トップレベルウィンドウが持っている config を呼び出して、menu というキーワード引数でそのメニューを渡せば、そのメニューがそのトップレベルウィンドウに取り付けられることになります。

次のプログラムは、プッシュボタンが押されるたびに、メニューバーが取り付けられたトップレベルウィンドウを生成します。

プログラムの例 menutoplevel.py

```
from tkinter import *

class MenuToplevel:
    def __init__(self):
        global count
        count += 1
        self.id = count
        toplevel = Toplevel()
        menubar = Menu(toplevel)
```

```
        menu = Menu(menubar, tearoff=0)
        menu.add_command(label='私について',
            command=self.aboutme)
        menubar.add_cascade(label='メニュー', menu=menu)
        toplevel.config(menu=menubar)
    def aboutme(self):
        print('私は' + str(self.id) +
            '番目のトップレベルウィンドウです。')

count = 0
root = Tk()
button = Button(root, text='トップレベルウィンドウの生成',
    command=MenuToplevel, font='Times 20')
button.pack()
root.mainloop()
```

17.3　オリジナルなダイアログボックス

17.3.1　オリジナルなダイアログボックスの基礎

第 16 章で説明したように、tkinter では、メソッドを呼び出すだけでさまざまなダイアログボックスを表示することができます。しかし、メソッドを呼び出すだけで表示することのできるダイアログボックスは、定型的なものに限られます。さて、それでは、定型的ではないダイアログボックスを表示したいときは、いったいどうすればいいのでしょうか。

ダイアログボックスというのはウィンドウの一種です。したがって、ダイアログボックスは、トップレベルウィンドウを生成することによって表示することができます。ただし、トップレベルウィンドウをダイアログボックスとして使う場合には、それが持っているグラブと破壊という機能が必要になります。

17.3.2　グラブ

ウィジェットに対する操作が終了するまでのあいだ、それ以外のウィジェットに対する操作を制限するという機能は、「グラブ」(grab) と呼ばれます。

グラブを使いたいときは、ウィジェットが持っている grab というメソッドを呼び出します。

グラブは、グラブを実行しているウィジェットが消滅すると、自動的に解除されます。

次のプログラムが生成するトップレベルウィンドウは、グラブを実行します。ですから、それが閉じられるまでのあいだ、ルートウィジェットに対しては何の操作もできなくなります。

プログラムの例　grab.py

```
from tkinter import *

class GrabbedToplevel():
    def __init__(self):
        toplevel = Toplevel()
        toplevel.grab_set()

root = Tk()
button = Button(root, text='トップレベルウィンドウの生成',
    command=GrabbedToplevel, font='Times 20')
button.pack()
root.mainloop()
```

17.3.3　破壊

ウィンドウには、自身を閉じるためのボタンが最初から備わっています。しかし、ダイアログボックスは、そのボタンとは別に、自身を閉じるプッシュボタンを持っているのが普通です。

ウィンドウを閉じるプッシュボタンがクリックされた場合には、ウィンドウを破壊するという動作を実行する必要があります。

すべてのウィジェットは、destroy というメソッドを持っています。これは、自分自身と、その子供になっているすべてのウィジェットを破壊するメソッドです。

次のプログラムが生成するトップレベルウィンドウは、「OK」というプッシュボタンをクリックすることによって閉じることができます。

プログラムの例　destroy.py

```
from tkinter import *

class ClosableToplevel():
    def __init__(self):
        self.toplevel = Toplevel()
        self.toplevel.grab_set()
        button = Button(self.toplevel, text='OK',
            command=self.close, font='Times 30')
        button.pack()
    def close(self):
        self.toplevel.destroy()

root = Tk()
button = Button(root, text=' トップレベルウィンドウの生成',
    command=ClosableToplevel, font='Times 20')
button.pack()
root.mainloop()
```

17.3.4　文字列を読み込むダイアログボックス

次のプログラムは、表示する文字列を変更するために、文字列を読み込むダイアログボックスを生成します。

プログラムの例　entrydialog.py

```
from tkinter import *

class EntryDialog:
    def __init__(self):
        self.toplevel = Toplevel()
        self.toplevel.grab_set()
        self.entry = Entry(self.toplevel, width=30,
            font='Times 20')
        self.entry.pack()
        self.entry.focus_set()
        button = Button(self.toplevel, text='OK',
            command=self.close, font='Times 30')
        button.pack()
    def close(self):
        var.set(self.entry.get())
        self.toplevel.destroy()

root = Tk()
var = StringVar()
var.set('string')
label = Label(root, textvariable=var, font='Times 20')
label.pack()
button = Button(root, text=' 文字列の変更',
    command=EntryDialog, font='Times 20')
button.pack()
root.mainloop()
```

第 18 章　標準ライブラリーのその他のモジュール

18.1　time

18.1.1　この章について

このチュートリアルでは、これまでに、Python の標準ライブラリーに含まれているモジュールとして、os、os.path、tkinter などを紹介してきましたが、Python の標準ライブラリーには、それら以外にもまだまだたくさんのモジュールが含まれています。

この章では、このチュートリアルにまだ登場していない標準ライブラリーのモジュールのうちで、よく使われるものをいくつか紹介したいと思います。

18.1.2　UNIX 時間

この章でまず最初に紹介するのは、時刻を扱う time というモジュールです。

time についての説明を始める前に、まず、その予備知識として、「UNIX 時間」(UNIX time) というものについて説明しておきたいと思います。

UNIX 時間というのは、UNIX を始めとする多くの OS で採用されている、時刻を表現する形式のひとつのことです。

UNIX 時間は、UNIX エポックと呼ばれる時刻から経過した秒数で、時刻を表現します。UNIX エポックは、1970 年 1 月 1 日午前 0 時 0 分 0 秒です。

第 9.5.2 項で説明したように、os というライブラリーの中で定義されている stat という関数は、引数としてパス名を受け取って、そのパス名で指定されたファイルまたはディレクトリの情報を持つオブジェクトを返します。そのオブジェクトが持っている最終アクセス時刻や最終更新時刻などの時刻の情報は、UNIX 時間で表現されています。

18.1.3　time で定義されている関数

time では、次のような関数が定義されています。

time	現在時刻を UNIX 時間で返す（クラスは float）。
gmtime	UNIX 時間を受け取って、その時刻を協定世界時 (UTC) であらわす struct_time クラスのオブジェクトを返す。引数を渡さなかった場合は現在時刻をあらわすオブジェクトを返す。
localtime	UNIX 時間を受け取って、その時刻を地方時であらわす struct_time クラスのオブジェクトを返す。引数を渡さなかった場合は現在時刻をあらわすオブジェクトを返す。
ctime	UNIX 時間を受け取って、その時刻を地方時であらわす、 `Wed Jul 10 11:35:54 2013` という形式の文字列を返す。引数を渡さなかった場合は現在時刻をあらわす文字列を返す。
asctime	struct_time クラスのオブジェクトを受け取って、その時刻を地方時であらわす文字列（形式は ctime と同じ）を返す。引数を渡さなかった場合は現在時刻をあらわす文字列を返す。

gmtime や localtime が返す、struct_time クラスのオブジェクトでは、次のような属性が、時刻を構成する要素に束縛されています。

tm_year	年（たとえば 2013）。
tm_mon	月（1 から 12 まで）。
tm_mday	日（1 から 31 まで）。
tm_yday	大晦日からの日数（元旦は 1）。
tm_wday	月曜日からの日数（月曜日は 0）。
tm_hour	時（0 から 23 まで）。
tm_min	分（0 から 59 まで）。
tm_sec	秒（0 から 59 まで）。
tm_isdst	夏時間フラグ。夏時間ならばプラス、そうでなければ 0、情報が得られなければマイナス。

次のプログラムの中で定義されている mtimelist という関数は、ディレクトリのパス名を受け取って、そのディレクトリの中にあるすべてのファイルについて、そのファイルの名前と最終更新時刻を出力します。

プログラムの例　mtimelist.py

```
from os import *
from os.path import *
from time import *

def mtimelist(dirname):
    for item in listdir(dirname):
```

```
        name = join(dirname, item)
        if isfile(name):
            st = stat(name)
            print(item + ': ' + ctime(st.st_mtime))
```

実行例

```
>>> from mtimelist import *
>>> mtimelist('/home/hoge')
umiushi.txt: Wed Jul 10 15:21:07 2013
namako.txt: Mon May 22 09:00:32 2006
kamenote.txt: Tue Aug 19 17:26:31 1997
```

18.2 sys

18.2.1 sys の基礎

この節では、sys という標準ライブラリーのモジュールについて説明したいと思います。

sys というモジュールで定義されているのは、Python のインタプリタに関連する関数などのオブジェクトです。それらのオブジェクトを使うことによって、たとえば、コマンドライン引数を取得したり、終了ステータスを指定してインタプリタを終了させたりすることができます。

18.2.2 コマンドライン引数

Python のインタプリタを起動するコマンドには、任意の個数の引数を書くことができます。Python のインタプリタを起動するコマンドに含まれている引数は、「コマンドライン引数」(command line argument) と呼ばれます。

sys は、argv という識別子を、コマンドライン引数から構成されるリストに束縛します。ただし、そのリストの 0 番目の要素は、起動したプログラムが格納されているファイルの名前です。

次のプログラムは、argv が束縛されているリストを出力します。

プログラムの例 argv.py

```
from sys import *
print(argv)
```

実行例

```
$ python argv.py namako umiushi hitode kamenote
['argv.py', 'namako', 'umiushi', 'hitode', 'kamenote']
```

18.2.3 プログラムの終了

sys で定義されている exit という関数は、インタプリタを終了させます。引数として整数を渡すと、それがインタプリタの終了ステータスになります。引数を省略した場合は 0 が終了ステータスになります。

整数以外のオブジェクトを引数として exit に渡すと、そのオブジェクトが標準エラー出力に出力されて、終了ステータスは 1 になります。

次のプログラムが表示するプッシュボタンは、クリックされると、exit を呼び出して、「私は exit に渡された引数です。」という文字列を引数としてそれに渡します。

プログラムの例 exit.py

```
from tkinter import *
from sys import *

def clicked():
    exit('私は exit に渡された引数です。')

root = Tk()
button = Button(root, text='exit の呼び出し',
    command=clicked, font='Times 20')
button.pack()
root.mainloop()
```

18.3 random

18.3.1 random の基礎

この節では、random という標準ライブラリーのモジュールについて説明したいと思います。

規則性を持たず、かつ、どの数値も同じ確率で出現する数値の列は、「乱数列」(random number sequence) または「乱数」(random number) と呼ばれます。

コンピュータというのは、何らかの規則に基づいて動作する機械ですから、厳密な意味での乱数を発生させることはできません。しかし、乱数に類似する数列をコンピュータに生成させることは可能で、そのような数列は、「擬似乱数列」(pseudorandom number sequence) または「擬似乱数」(pseudorandom number) と呼ばれます。

Python では、random というモジュールで定義されている関数を呼び出すことによって、擬似乱数を発生させることができます。

18.3.2 浮動小数点数の擬似乱数

random というモジュールで定義されている random という関数は、0.0 から 1.0 までの範囲内にあるランダムな浮動小数点数を戻り値として返します。

次のプログラムは、0.0 から 1.0 までの範囲内にある 5 個の浮動小数点数から構成される擬似乱数を発生させます。

プログラムの例　randomfloat.py

```
from random import *
for i in range(5):
    print(random())
```

実行例

```
$ python randomfloat.py
0.17144371292907223
0.14570298210501298
0.05991105114391182
0.05045412242242564
0.9777900877759715
```

18.3.3 整数の擬似乱数

random で定義されている randrange という関数は、ランダムな整数を戻り値として返します。

randrange は、引数として 1 個または 2 個または 3 個の整数を受け取ります。戻り値は、引数が 1 個の場合は 0 から引数までの範囲内にあるランダムな整数で、引数が 2 個の場合は 1 個目から 2 個目までの範囲内にあるランダムな整数で、引数が 3 個の場合は 1 個目から 2 個目までの範囲内にある 3 個目ずつ離れたランダムな整数です。

次のプログラムは、300 から 700 までの範囲内にある 10 ずつ離れた 5 個の整数から構成される擬似乱数を発生させます。

プログラムの例　randrange.py

```
from random import *
for i in range(5):
    print(randrange(300, 700, 10))
```

実行例

```
$ python randrange.py
470
660
310
330
440
```

参考文献

[Ceder,2010] Naomi R. Ceder, *The Quick Python Book, Second Edition*, Manning Publications, 2010, ISBN 978-1-935182-20-7. 邦訳（新丈径）、『空飛ぶ Python 即時開発指南書』、翔泳社、2013、ISBN 978-4-7981-3080-4。

[Downey,2013] Allen Downey, *Think Python: How to Think Like a Computer Scientist, Version 2.0.6*, Green Tea Press, 2013. 原書 *Version 2.0.3* の邦訳（相川利樹）、『Think Python：コンピュータサイエンティストのように考えてみよう・第 0.90 版』、2013。

[Language,2013] Guido van Rossum, Fred L. Drake, Jr. (ed.), "The Python Language Reference, Release 3.3.1", Python Software Foundation, 2013.

[Library,2013] Guido van Rossum, Fred L. Drake, Jr. (ed.), "The Python Library Reference, Release 3.3.1", Python Software Foundation, 2013.

[Lundh,1999] Fredrik Lundh, "An Introduction to Tkinter", 1999.

[Shipman,2013] John W. Shipman, "Tkinter 8.5 Reference: A GUI for Python", New Mexico Institute of Mining and Technology, 2013.

[Tutorial,2013] Guido van Rossum, Fred L. Drake, Jr. (ed.), "Python Tutorial, Release 3.3.1", Python Software Foundation, 2013.

[柴田,2012] 柴田淳、『みんなの Python・第 3 版』、ソフトバンククリエイティブ、2012、ISBN 978-4-7973-7159-8。

索引