@IT eBookシリーズ　*Vol.33*

機械学習で注目のPythonを学ぶ Visual Studioで始める Pythonプログラミング入門

Insider.NET編集部 かわさきしんじ[著]

@IT eBookシリーズ　*Vol.33*

機械学習で注目のPythonを学ぶ Visual Studioで始める Pythonプログラミング入門

機械学習分野で注目が集まっているプログラミング言語「Python」を、.NET プログラマーになじみの Visual Studio で始めてみよう。Python Tools for Visual Studio を使うと、高度な開発者支援機能を活用して、Python プログラミングが行える。

特集：Visual Studio で始める Python プログラミング

特集：Python 3.6 の新機能

初出：@ IT Insider.NET フォーラム
・特集：Visual Studio で始める Python プログラミング
・特集：Python 3.6 の新機能
【注意】
・eBook 化に際して、上記の記事を一部加筆修正しています。

特集：Visual Studio で始める Python プログラミング

1. Python Tools for Visual Studio のセットアップ

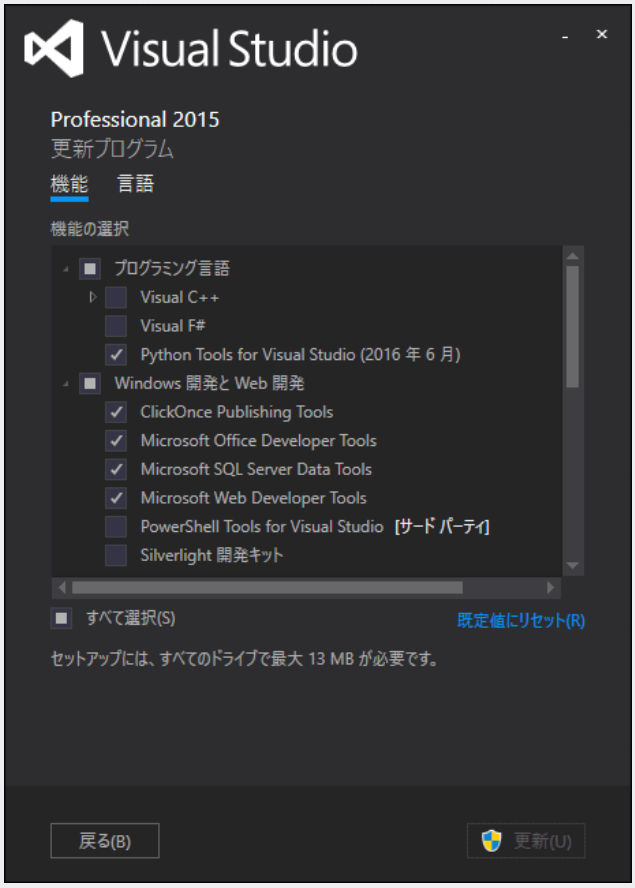
Python Tools for Visual Studio を使うと、高度な開発者支援機能を活用して、Python プログラミングが行える。まずはその概要について見ていこう。

Visual Studio（以下、VS）が .NET 開発者だけのものだった時代はもう終わっている。現在では、.NET 言語に加えて、JavaScript や TypeScript など、さまざまな言語がサポートされるようになった。本電子書籍では、そうした言語の中でも Python に焦点を当て、VS で Python プログラミングを始めるための基礎知識を紹介していこう。

Python Tools for Visual Studio

VS 2015 以前で Python を利用するには「Python Tools for Visual Studio」（以下、PTVS）が必要になる。PTVS は、マイクロソフトとコミュニティーによりオープンソースソフトウェア（OSS）として開発が行われている VS 用の Python プラグインである。2016 年 9 月 8 日時点の最新バージョンは 2.2.5 で、サポートしている VS のバージョンは VS 2015 のみとなっている（PTVS 2.2.2 では VS 2013 が、PTVS 2.1.1 では VS 2010 ／ 2012 がサポートされている）。VS 2017 では［Python 開発］ワークロードをインストールすることで、Python がサポートされる。これについては次章で取り上げよう。

PTVS のインストールには、VS のインストーラーで［プログラミング言語］の下にある［Python Tools for Visual Studio］にチェックを入れるか、PTVS のリリースノートページなどからインストーラーをダウンロードして実行する。



VS 2015 で PTVS をインストールするように構成を行っているところ

PTVS とは別に Python の処理系をインストールする必要もある。PTVS はその名の通り、VS 用の Python サポートツールであって、Python 自体は含まれていない。逆に Python 処理系を含んでいないことから、PC へ複数の Python 処理系をインストールして、それらを VS 環境内で簡単に切り替えられるようになっている。多くの処理系は、それらをインストールすることで PTVS が自動的に認識してくれる。例えば、「PTVS Installation」ページでは以下の処理系などが挙げられている。

- CPython
- IronPython
- Anaconda

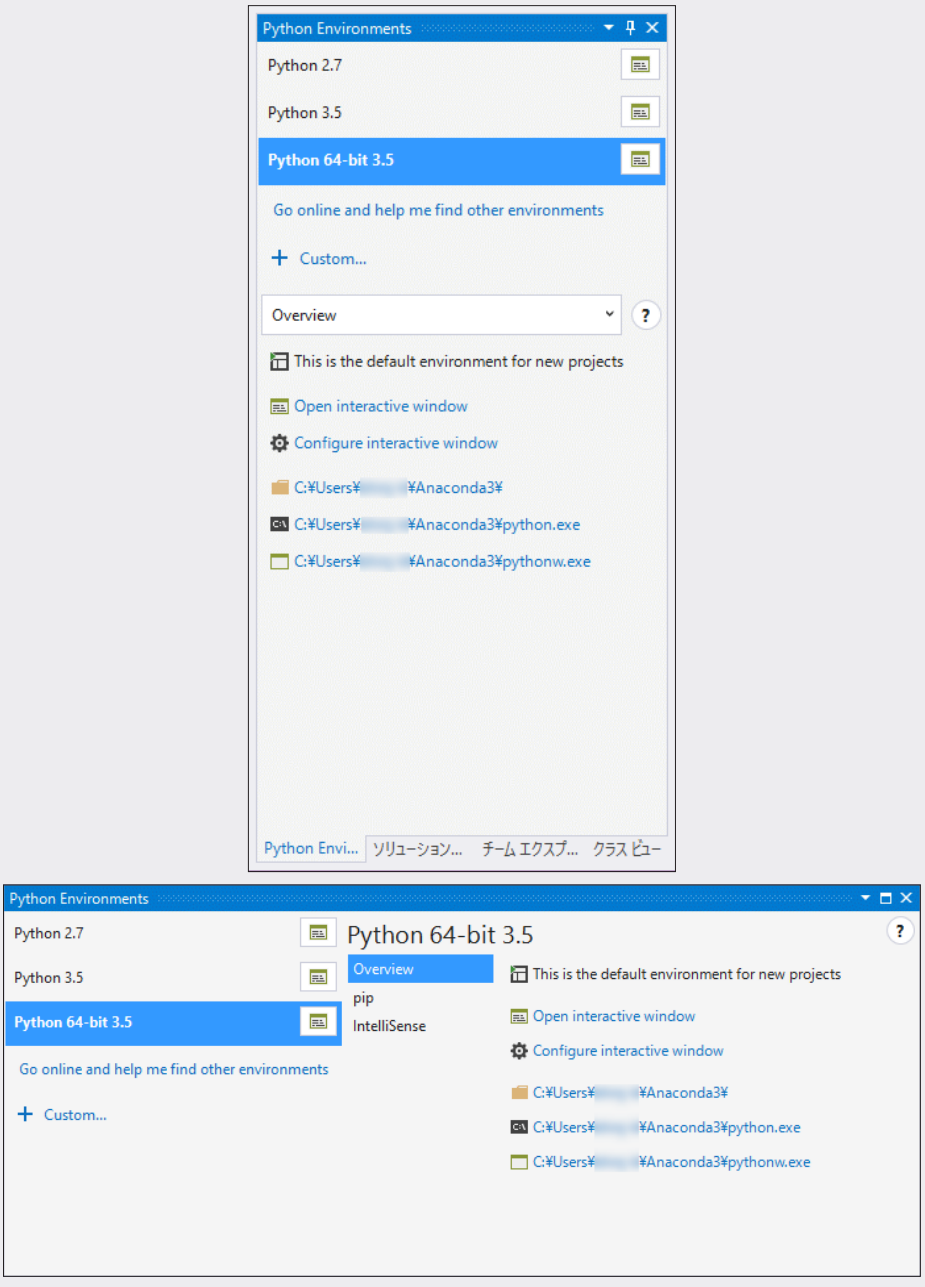
Anaconda はデータサイエンス／データ解析系の技術者向けに Python 処理系に加えて多くのライブラリ／フレームワークをあらかじめパッケージングしてあるものだ。Python のパッケージのインストールで苦労したくないのであれば、これらを最初から使ってしまうのも 1 つの手だ。

この他にも Jython（Python の JVM 実装）なども PTVS ではサポートされているが、ドキュメントによるとデバッグなどの機能は使えない。

本書では CPython 2.7 ／ 3.5（32bit）と Anaconda 4.1.1（Python 3.5 ／ 64bit）を例としてインストールした（インストール手順は省略する）。

[Python Environments] ウィンドウ

PTVS および各種の Python 処理系をインストールすることで、VS で Python プログラミングが可能になる。PTVS はインストールされている Python 処理系を（多くの場合）自動認識してくれ、どれをデフォルトの環境とするかを指定したり、それぞれの環境の構成を行ったりできる。これを行うための入口となるのが［Python Environments］ウィンドウだ。



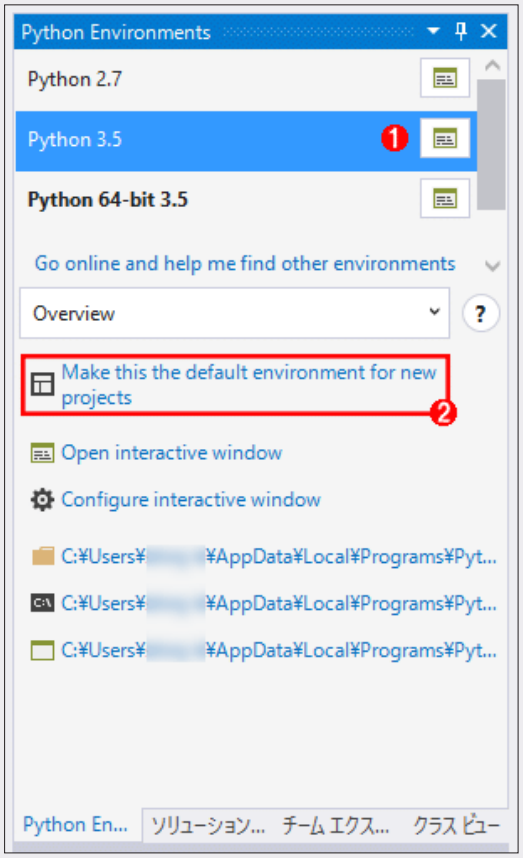
［Python Environments］ウィンドウ

上はソリューションエクスプローラーなどと同じ位置にドッキングしたもの。下はフロートウィンドウとして表示して、サイズを変更したもの。

→目次に戻る

3 つの Python 処理系（環境）が認識されていることが分かる。そして、ボールド書体で表示されているものが現在デフォルトで使用される環境となる（ここでは［Python 64-bit 3.5］＝Anaconda 4.1.1 になっている。下に「This is the default environment for new projects」と表示されていることに注意）。

デフォルトの環境を変更するには、対象の環境を選択して、その下にある［Make this default environment for new projects］をクリックする。また、［Go online and help me find other environments］をクリックすると、「Selecting and Installing Python Interpreters」ページが VS 内で表示される（どんな処理系をインストールできるかを知りたい人は英語のドキュメントだが一読しておこう）。



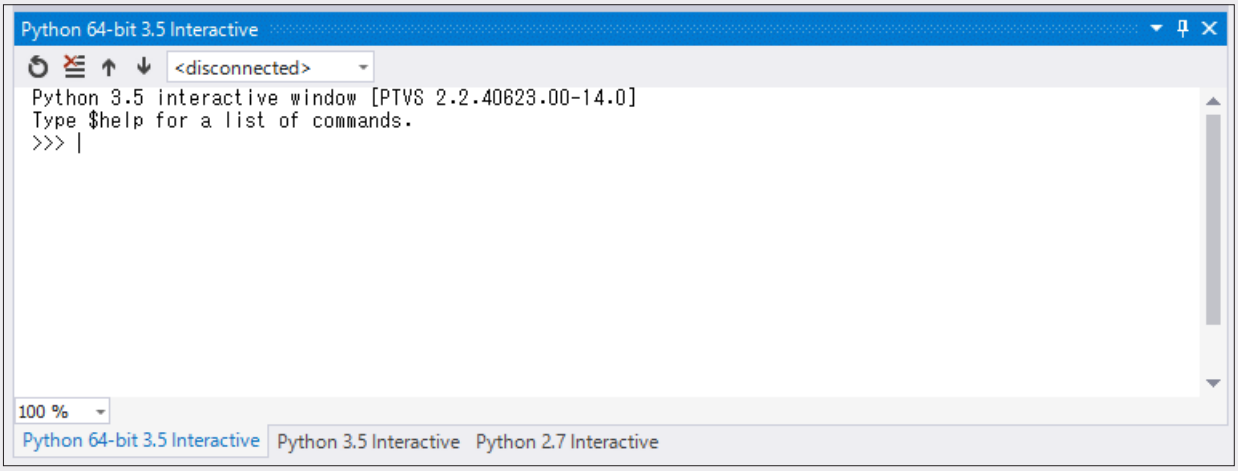
デフォルトの環境を切り替える

デフォルトの環境にしたいものを選択して（①）、［Make this the default environment for new projects］をクリックする（②）。

［Python Environments］ウィンドウでは各種の構成も行えるが、以下では［Open interactive window］をクリックして、VS 内で Python を対話的に使ってみよう。

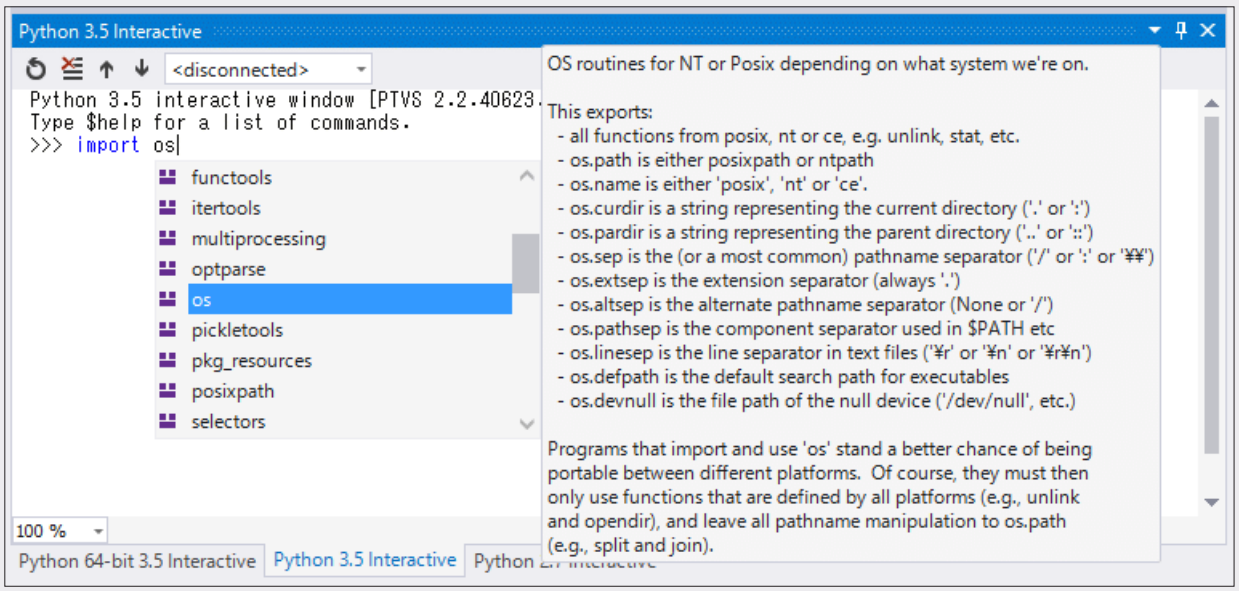
[Interactive] ウィンドウを使ってみる

[Open interactive window] をクリックすると、VS 内に [Interactive] ウィンドウが表示される。インストールされている環境ごとに別々のタブを開くことも可能だ。以下に例を示す。



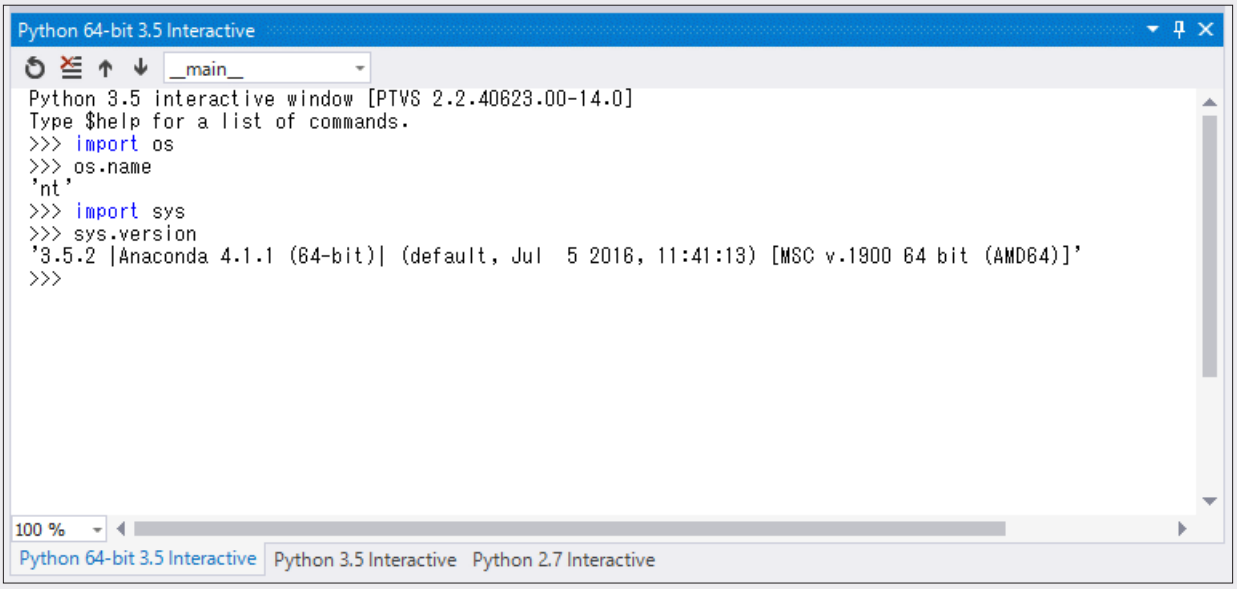
[Interactive] ウィンドウ（3 つのタブがあることに注意）

まずは Python にデフォルトで組み込まれているモジュール「os」をインポートしてみよう。「import os」と入力すると、IntelliSense が働いて、概要が表示される。



PTVS では IntelliSense が働く

os モジュールは OS 依存の機能に対する統一的なインタフェースを提供する。そこで次に「os.name」と入力してみよう。Windows で実行している場合、その結果は「'nt'」になる（Windows NT の「nt」）。同様にして sys モジュールをインポートして、今度は「sys.version」と入力してみる（sys モジュールは Python インタープリタ=［Interactive］ウィンドウに関連するシステムパラメーターや関数を管理するモジュール）。以下は「Python 64-bit 3.5」環境での実行結果だ。



sys.version の値を表示

os.name の値は今回インストールしている 3 つの環境全てで「'nt'」になるが、sys.version の値は環境ごとに異なる（興味のある方は、ご自分でも幾つかの環境をインストールして実行してみよう）。このように、PTVS は環境ごとに独自のコンテキストを持つ［Interactive］ウィンドウを通して Python の対話的実行が可能になっている。

モジュールをインポートして、そこで定められている値を表示するだけではつまらないので、今度は for 文とリストを使ってみよう。以下に例を示す。

```
>>> lst = list(range(3))
>>> lst
[0, 1, 2]
>>> for n in lst:
...     print(n * 2)
...
0
2
4
```

リストの要素を 2 倍にして出力

最初の行では「0 ～ 2 の範囲」を表す range 型のオブジェクトを作成して、それを基にリストオブジェクトを作成している。そして、リストを反復して、各要素の値を 2 倍してコンソールに出力している。

関数も定義できる。これについては説明の必要もないだろう。

```
>>> def hello(to):
...     print("hello " + to)
...
>>> hello("Insider.net")
hello Insider.net
```

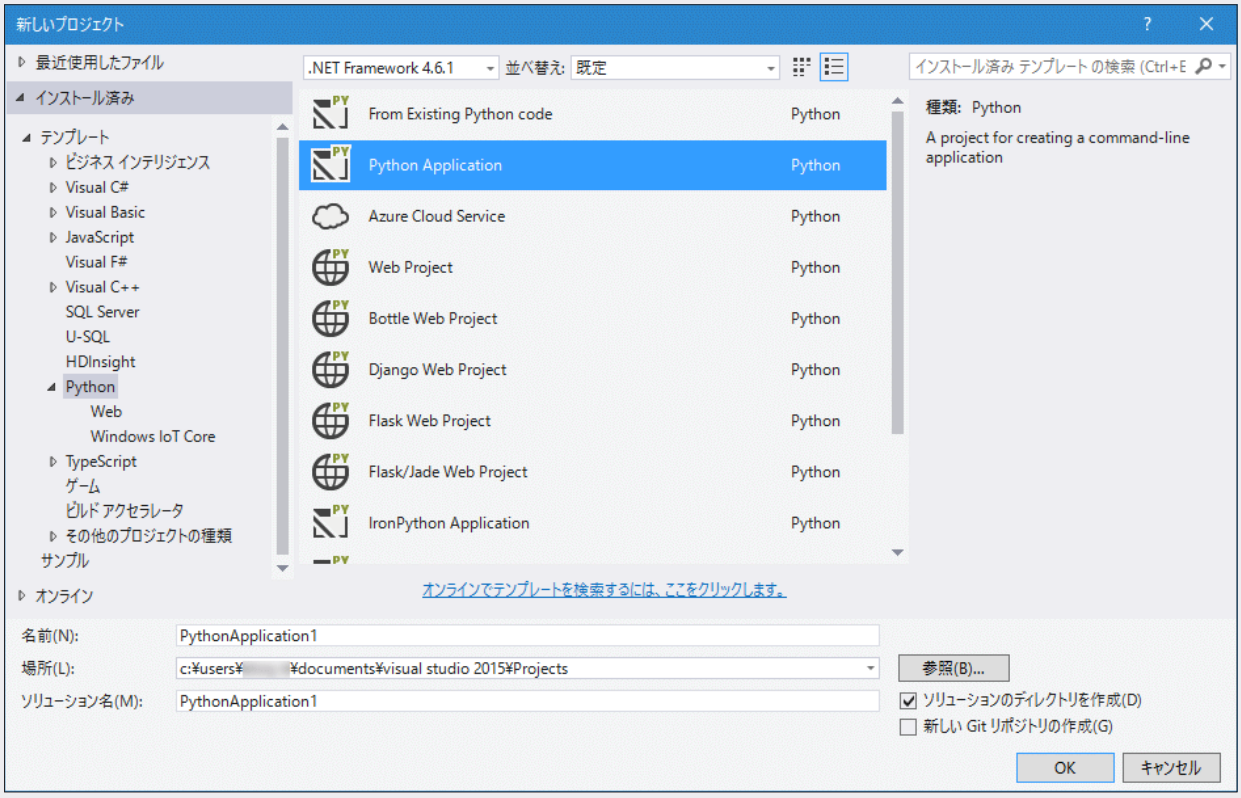
関数の定義と呼び出し

キーワードで始める各文の末尾にあるコロン（:）とインデントによるブロックの指定は Python コードの見た目における特徴的な要素といえる（Python の文法については 3 章以降で見ていこう）。

ここまでは PTVS をインストールした VS の中で特にプロジェクトを作成することもなく、Python の対話的実行を試してきた。次は Python のコンソールアプリプロジェクトを作成して、これを実行してみよう。

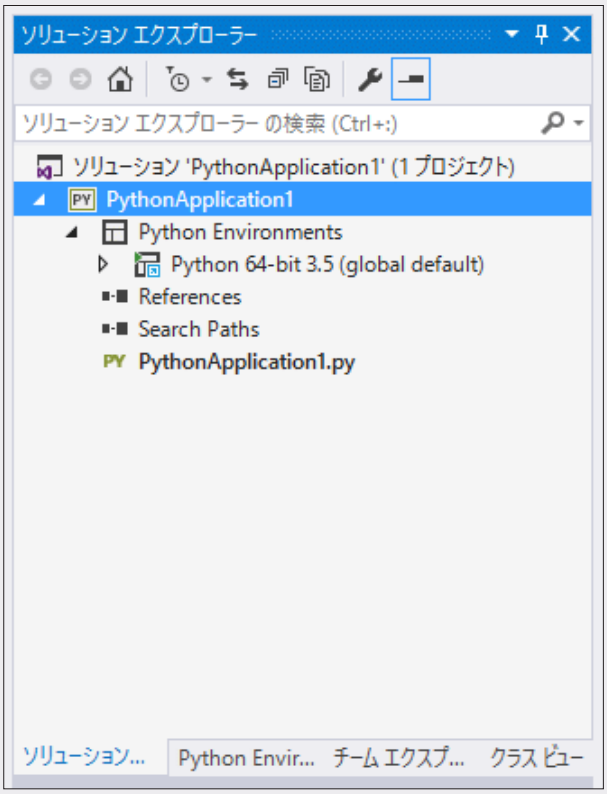
Python コンソールアプリ

Python のコンソールアプリプロジェクトを新規に作成するといっても、通常の VS の操作と変わらない。［新しいプロジェクト］ダイアログでテンプレートを指定するだけだ。今回は［Python Application］を選択している。画像を見れば分かる通り、PTVS をインストールすると、Python で Web アプリを開発するためのテンプレートが利用できるようになる。



［新しいプロジェクト］ダイアログ

作成直後のソリューションエクスプローラーを以下に示す。



Python のコンソールアプリプロジェクト作成直後のソリューションエクスプローラー

特徴的なのは［Python Environments］という項目だ。ここには現在のデフォルトの Python 環境が表示される（切り替えるには、上述したように［Python Environments］ウィンドウを使用する）。［Search Paths］には Python が検索するパスや ZIP アーカイブを指定できるが、本書では利用しない。プロジェクトで特定のディレクトリや ZIP アーカイブ中のファイルを参照したい場合にこれを利用する。

詳細は割愛するが、以下では「mymodule」というモジュールを新規に作成し（mymodule.py ファイル）、そこで定義されている 2 つの関数を PythonApplication1.py ファイルでインポートしている。

mymodule.py ファイルの内容は次の通りだ。

```
def fact(n):
  if (n <= 1):
    return 1
  else:
    return n * fact(n-1)

def fib(n):
  if (n < 0):
    return -1
  elif(n == 0):
    return 0
  elif (n == 1):
    return 1
  else:
    return fib(n-2) + fib(n-1)
```

階乗とフィボナッチ数を求める関数を定義しているモジュール（mymodule.py ファイル）

やっていることは単純なのでコードの説明は割愛する。PythonApplication1.py ファイルのコードは次の通りだ。

```
import sys
from mymodule import fact, fib

print(sys.version)
print("factorial of 3 is {0}".format(fact(3)))

for n in range(7):
  print("fib({0}) is {1}".format(n,fib(n)))
```

mymodule モジュールから fact 関数と fib 関数をインポート（PythonApplication1.py ファイル）

mymodule モジュール（の 2 つの関数）に加えて、先ほど使った sys モジュールもインポートしている（「from モジュール import 識別子」によるインポートを行うことで、モジュール名を省略してインポートした識別子を利用できていることに注目）。「format」メソッドは出力の書式化を行うために使用している。Python では「"sum: " + 100」のようにして文字列と数値を「暗黙的」に連結することは許されていないために、このように書式化を行うか、str(...) のようにして文字列を取得してから連結を行う必要がある。なお、最近はやりの文字列補間機構は Python 3.6 でサポートされた。これについては「Python 3.6 で追加された新機能をザックリ理解」で取り上げる。

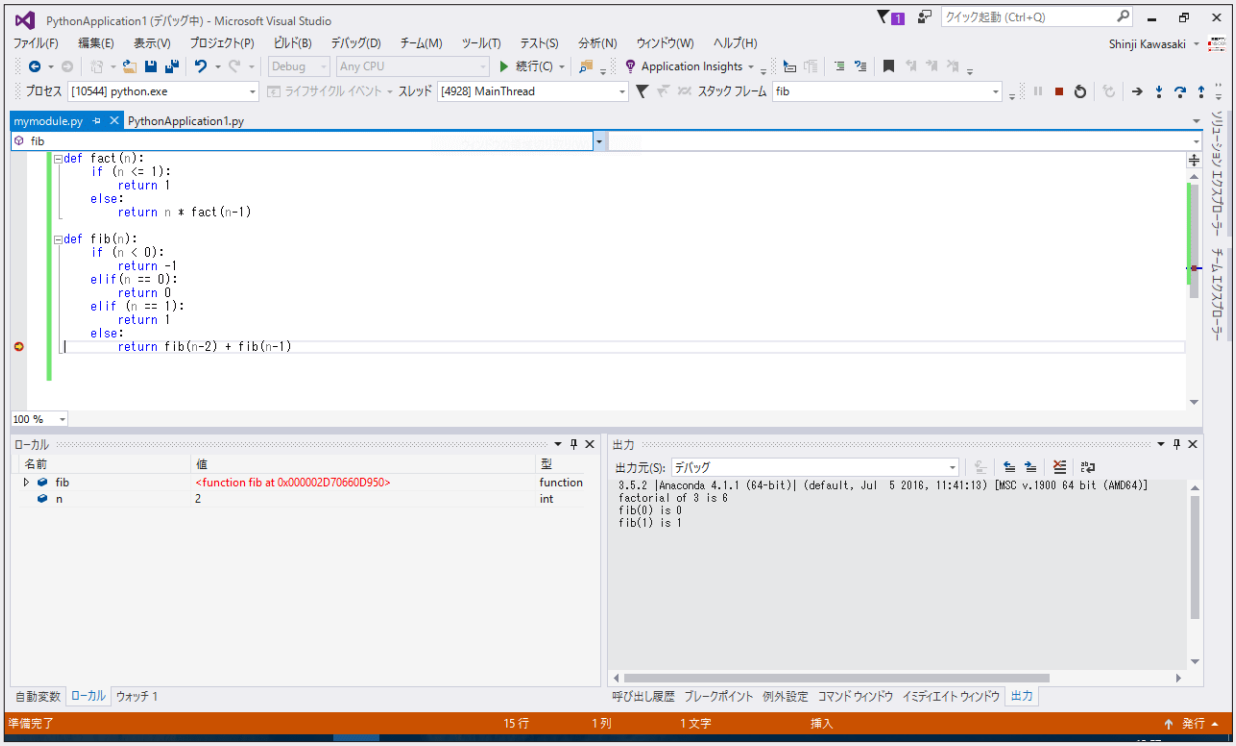
これをデバッグ実行すると次のような結果になる。

```
3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul  5 2016, 11:41:13) [MSC v.1900 64 bit
(AMD64)]
factorial of 3 is 6
fib(0) is 0
fib(1) is 1
fib(2) is 1
fib(3) is 2
fib(4) is 3
fib(5) is 5
fib(6) is 8
```

実行結果

先ほど見たように、[Python Environment] が [Python 64-bit 3.5] になっているので、「print(sys.version)」呼び出しの結果が Anaconda のものになっている。デフォルトの Python 環境を変更すれば、この結果は異なるものになる。

開発環境が VS である以上、ブレークポイントを設定すれば、ステップ実行も可能だ。

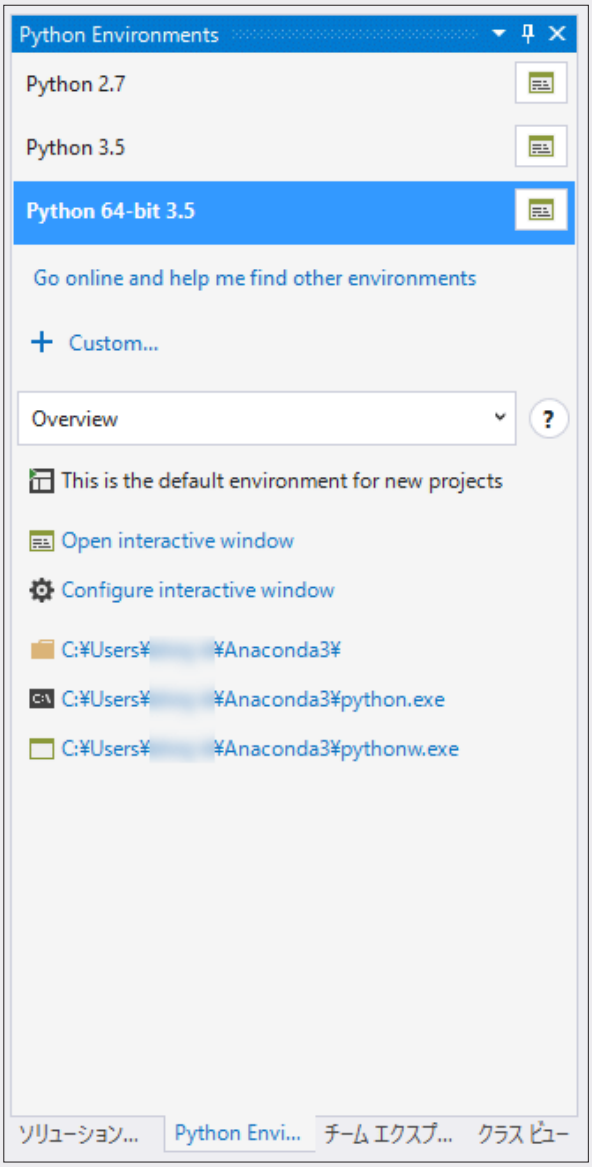


ブレークポイントでプログラムの実行が中断されたところ

VS が誇る IntelliSense や高度なデバッグ機構を Python プログラミングにおいても活用できるのはうれしいところだ。

環境の構成

［Python Environments］ウィンドウのドロップダウン（横長表示の場合は中央のペーン）には構成／一覧表示が可能な項目が表示される。デフォルトでは［Overview］が選択されており、ウィンドウ上部にあるリストボックスで選択している環境の概要が表示される。

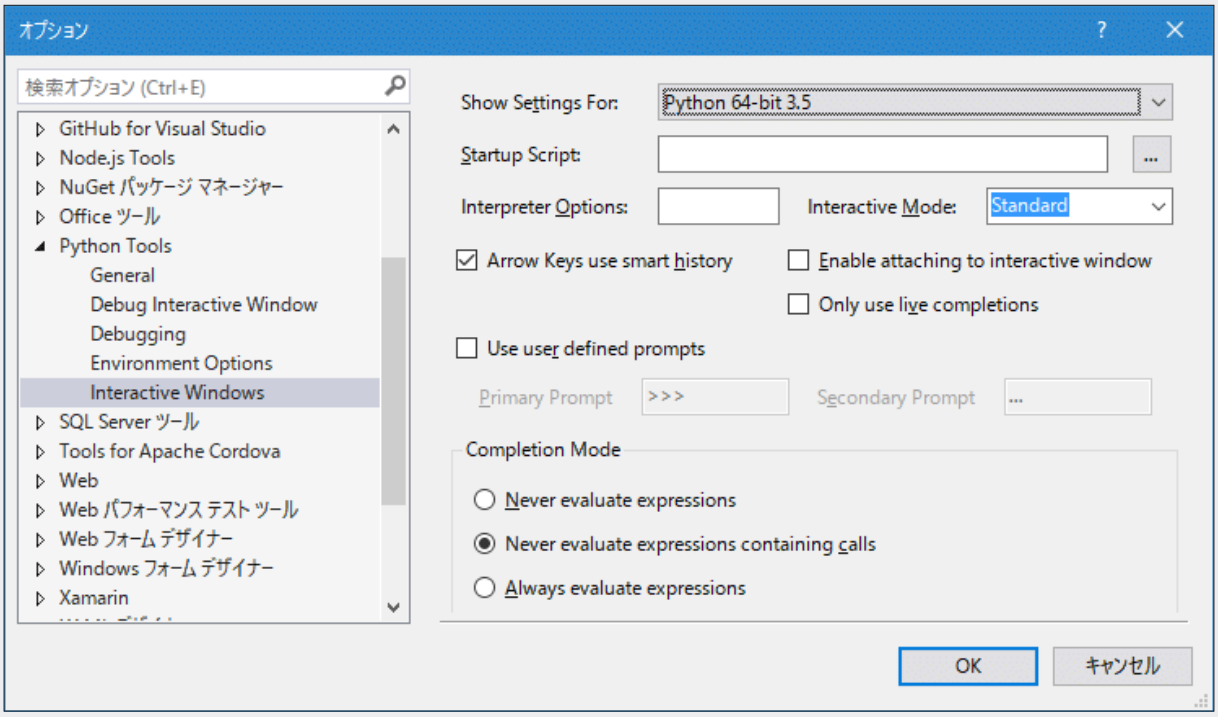


選択した環境の概要

［Overview］ページに表示されるのは以下のものだ。

- デフォルトの環境かどうか
- ［Interactive］ウィンドウを開く
- ［Interactive］ウィンドウの構成画面を開く
- 環境のインストール先ディレクトリ／ python コマンド／ pythonw コマンドのパス

［Interactive］ウィンドウの表示については上で試した通りだ。［Configure interactive window］をクリックすると、［オプション］ダイアログに［Interactive］ウィンドウの構成画面が表示される。

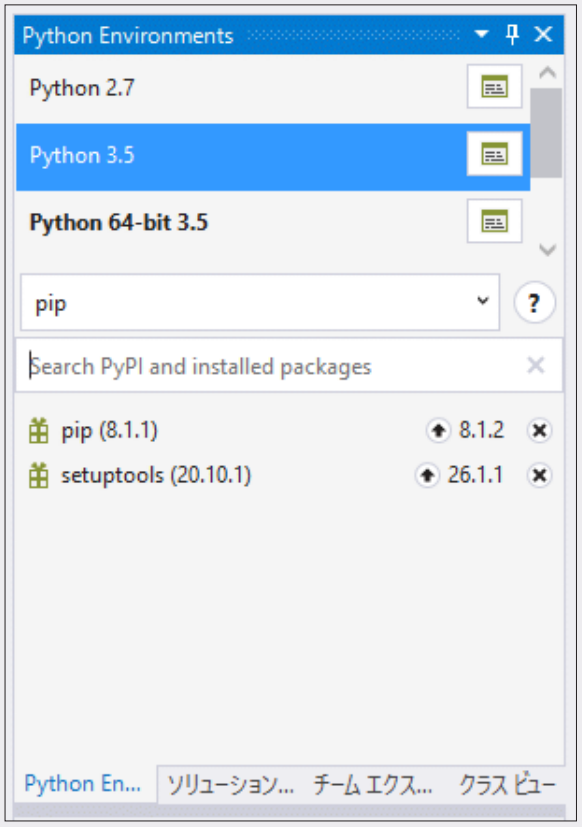


［オプション］ダイアログ

ここでは［Interactive］ウィンドウ起動時に実行するスクリプトの指定、ウィンドウのモードの指定、その他の構成が行える。後者の「モード」というのは標準の［Interactive］ウィンドウを使用するか、IPython と呼ばれるより高度な機能を持つ Python シェルを利用するかなどを指定可能だ。

インストール先のディレクトリと python コマンドのリンクをクリックすると、フォルダエクスプローラー／コマンドプロンプトが表示されるので、ディレクトリをブラウズしたり、直接コマンドプロンプトから python コマンドを実行したりしたい場合に便利に使える。

ドロップダウンから［pip］を選択すると、その環境にインストールされているパッケージが一覧表示される（pip は Python 用の代表的なパッケージ管理ツール）。検索ボックスに検索語を入力すれば、PyPI（the Python Package Index。Python 用のパッケージのリポジトリ）でのパッケージの検索とインストールも可能だ。

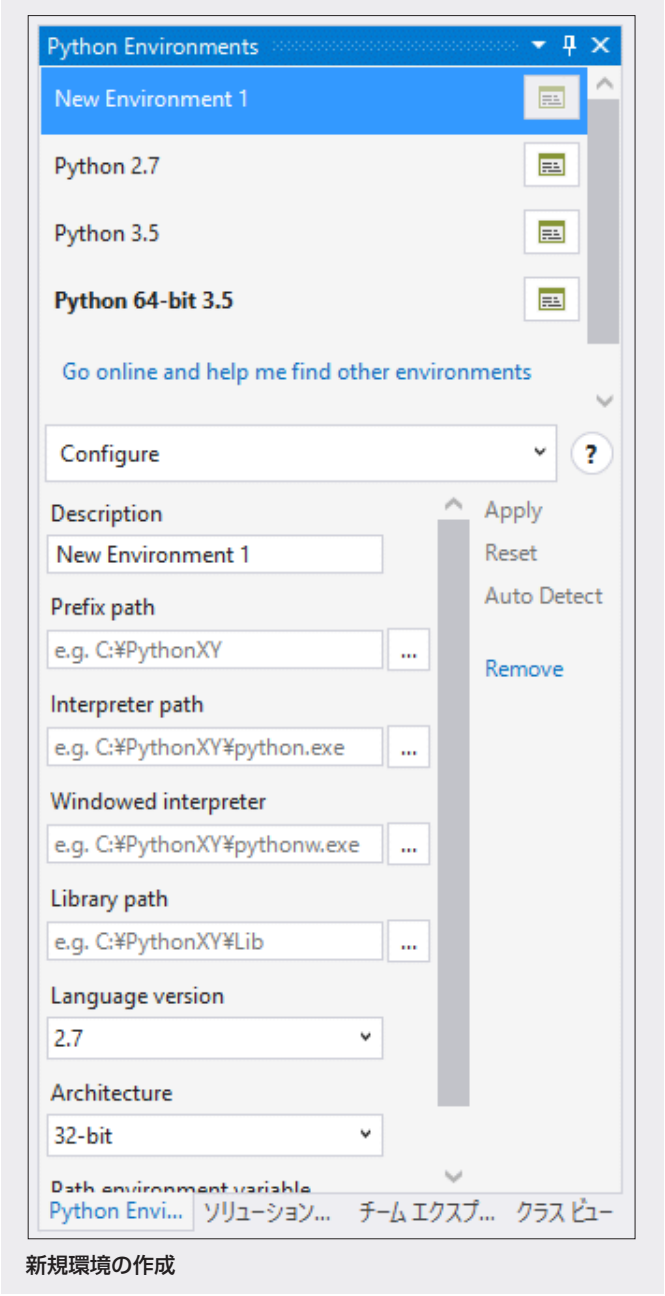


選択した環境にインストールされているパッケージの表示

ここでは［Python 3.5］環境にインストールされているパッケージが表示されている。［Python 64-bit 3.5］（Anaconda）ではこれよりもはるかに多くのパッケージが標準でインストールされている。

ドロップダウンから［IntelliSense］を選択すると、IntelliSense が認識している Python モジュールが表示される。

インストールしている Python 処理系が認識されていない、あるいは独自の環境設定を行いたい場合には［+ Custom］をクリックする。これにより、新しい環境の作成ページが表示される。



新規環境の作成

◇ ◇ ◇ ◇ ◇ ◇

本章では PTVS の概要と［Interactive］ウィンドウによる Python の対話的実行、簡単なアプリの作成と実行、PTVS の設定について見てきた。次章では VS 2017 における［Python 開発］ワークロードを使った、Python サポートのインストール方法などを取り上げる。Python の文法の基本に進みたい方は「Python の文法、基礎の基礎」まで飛ばしてくださって構わない。

特集：Visual Studio で始める Python プログラミング

2. Visual Studio 2017 における Python サポート

［Python 開発］ワークロードのインストール方法、［Python 環境］ウィンドウ、［Interactive］ウィンドウ、ソリューションエクスプローラーの構成などを説明する。

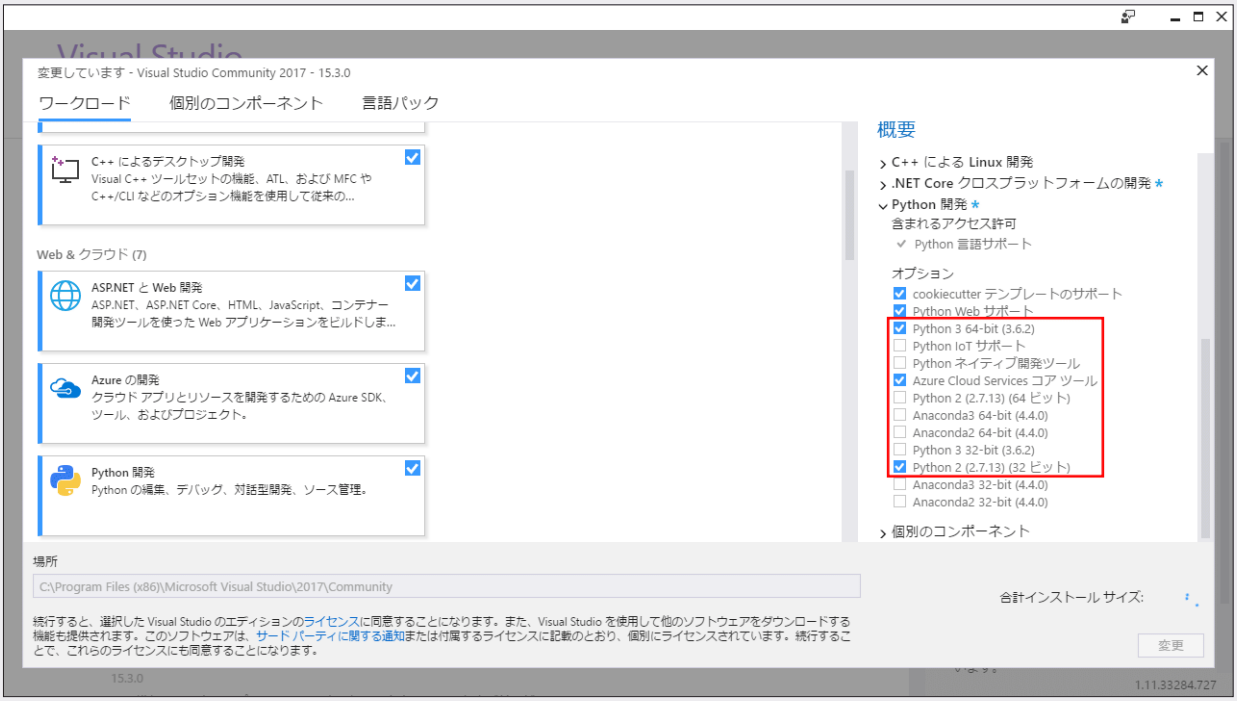
VS 2017 でインストーラーの方式が変更されたことで、Python サポート機能は、第 1 章で見た Python Tools for Visual Studio という形式ではなく、「ワークロード」という形でインストールされるようになった。以下では、そのインストール方法と VS 2017 の IDE と Python がどのような形で統合されているかを見ていく。

［Python 開発］ワークロード

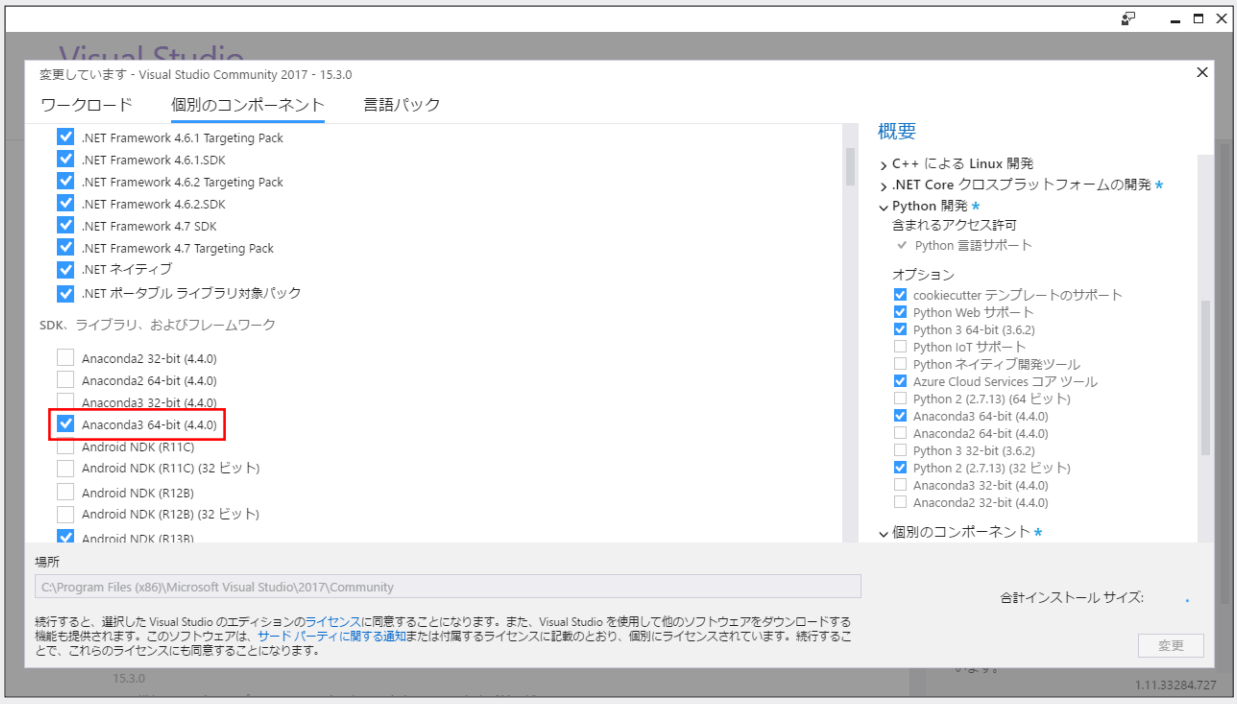
VS 2017 で Python を利用するには、［Python 開発］ワークロードを VS Installer で選択する。

VS Installer で［Python 開発］ワークロードを選択（赤枠内）

このとき VS Installer の右側に常に表示される［概要］ペーンあるいは［個別のコンポーネント］タブで、［Python 開発］ワークロードと同時にインストールする Python のバージョンやツール類などを選択できる（従来の Python Tools for VS では Python 処理系を別途インストールする必要があったが、［Python 開発］ワークロードでは同時にインストールが可能となっている）。複数の処理系をインストールした場合は、どの処理系を使用するかなどを、後述する［Python 環境］ウィンドウで切り替えられる。



［概要］ペーンでは、同時にインストールする Python のバージョンやツール類を選択できる



［個別のコンポーネント］タブでも、同時にインストールする Python のバージョンやツール類を選択できる

上は［ワークロード］ページ右側の［概要］ペーンで Python 2.7 ／ 3.6 をインストールするように選択をしているところ（赤枠内）。下は、［個別のコンポーネント］タブでさらに Anaconda 3 もインストールするようにしたところ（赤枠内）。

ただし、Python 処理系はこのワークロードとは独立して、python.org などからダウンロード、インストールしても構わない（その場合の注意点を最後に記してあるので、興味のある方はそちらも参照してほしい）。VS 2017 は基本的にはインストール済みの Python を自動的に認識して、それを利用できるようにしてくれる。

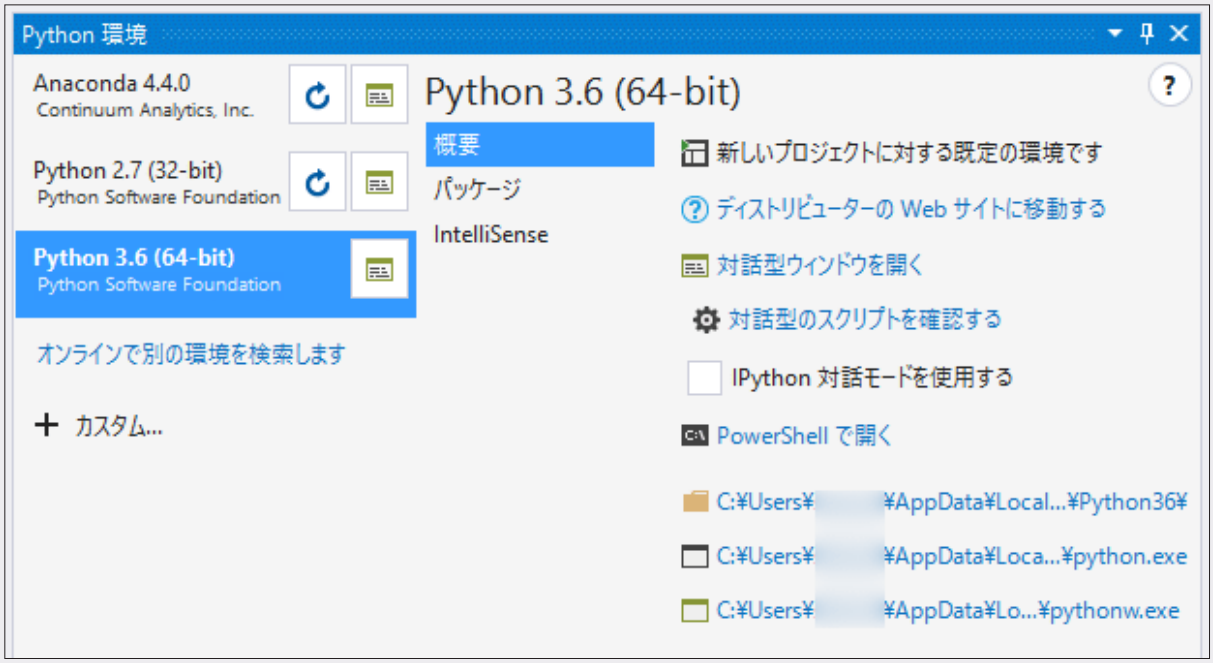
基本的にはこれだけで、VS 2017 で Python プログラミングが可能になる。次に、VS 2017 の IDE に含まれる Python サポートを見てみよう。ここでは、幾つかの Python 処理系（Python 2.7 ／ 3.6、Anaconda 3）とデフォルトで選択されるコンポーネント（Python 言語サポート／ Python Web サポートなど）をインストールしたものとして話を進める。また、以下では VS 2017 Update 3（15.3）で動作を確認している。

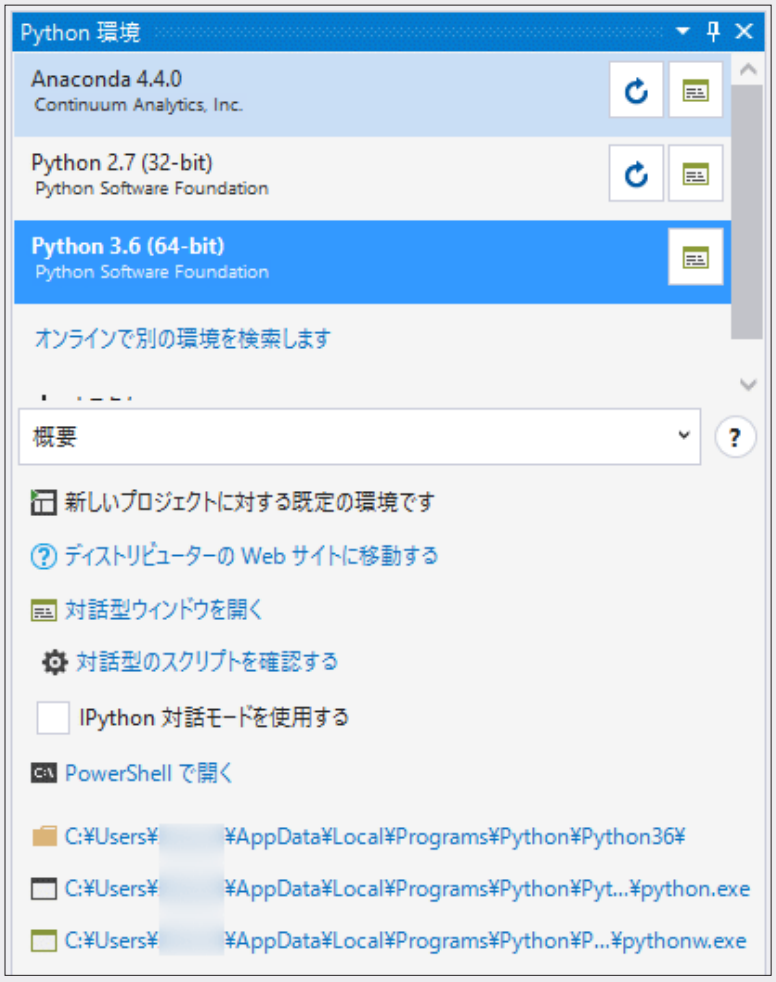
VS 2017 IDE における Python サポート

［Python 開発］ワークロードをインストールすると、Python 用のプロジェクトテンプレート、［Python 環境］ウィンドウ、対話型ウィンドウ（ここでは、実際のウィンドウの表示に合わせて、これを［Interactive］ウィンドウと表記する）、IntelliSense、デバッグサポートなどの機能がインストールされる。その中から、幾つかをここで紹介していこう。

［Python 環境］ウィンドウ

このウィンドウは VS 2017 で使用する Python 処理系に関する構成などを行うためのものだ。Python アプリのプロジェクトで使用するデフォルトの環境がどれかを指定したり、個々の環境に関してさまざまな設定を行ったりできる。メニューバーの［表示］－［その他のウィンドウ］－［Python 環境］を選択することで、その表示／非表示を切り替えられる。





［Python 環境］ウィンドウ

上は横に広く表示領域をとったもので、下はソリューションエクスプローラーなどと同様な表示としたもの。

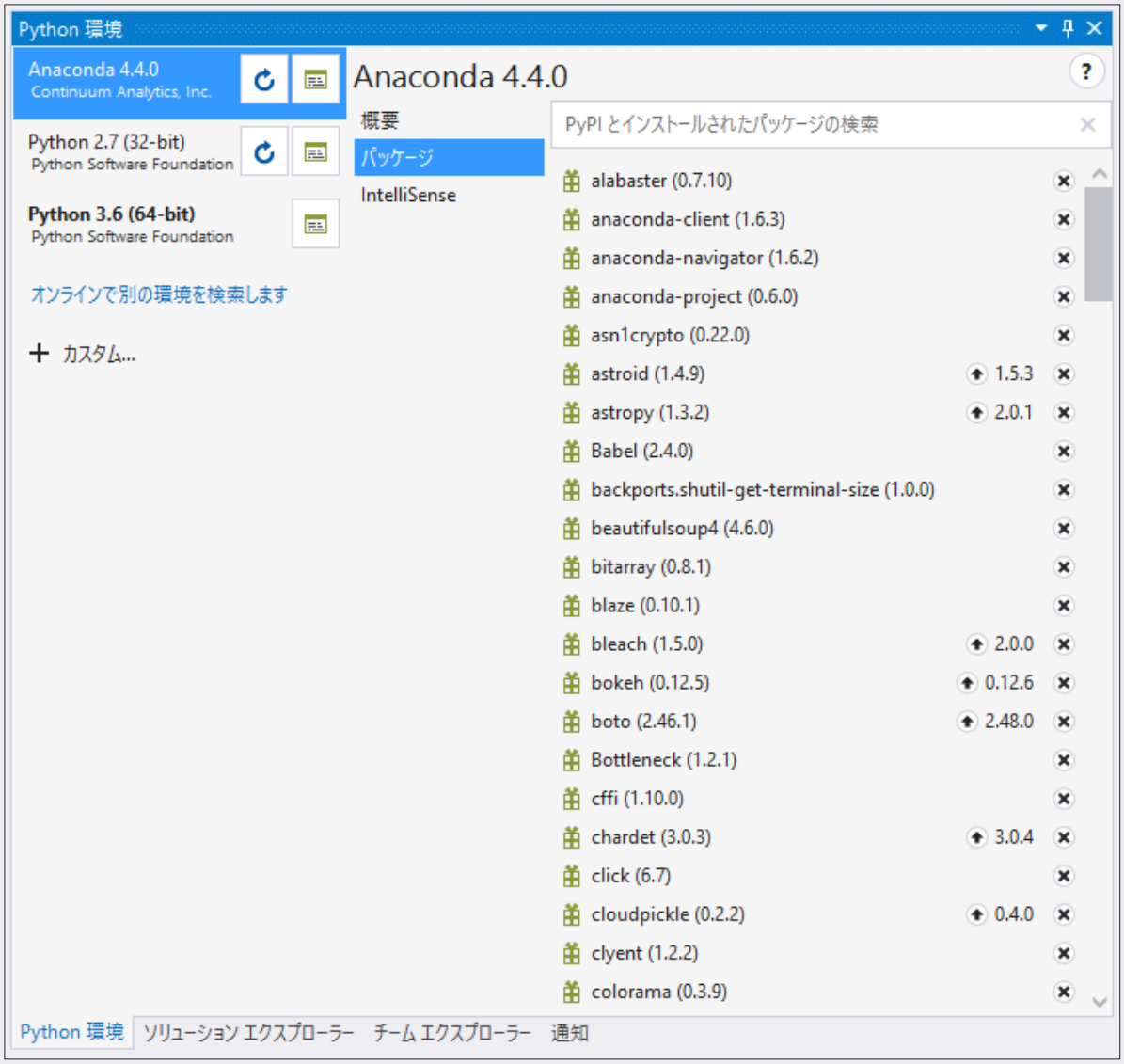
→目次に戻る

上の画面を見ると分かるように、ここでは Anaconda 4.4.0、Python 2.7、Python 3.6 の 3 種類の環境がインストールされ、VS 2017 で使用できるようになっている。そして、環境ごとに［概要］［パッケージ］［IntelliSense］の 3 つのタブがあり、それぞれの構成の一覧や変更が行える。別の処理系をインストールするには［オンラインで別の環境を探します］リンクをクリックする。［カスタム］というのは、既存の環境に対して、独自の構成を行う際に利用する（既存の処理系を、VS 2017 が自動的に発見できなかった場合などに利用する）。

上の画面で［Anaconda 4.4.0］と［Python 2.7］にあるリロードボタンは、IntelliSense で利用する補完データベースが最新ではないので、これを更新しろという意味だ。クリックすると、データベースの更新が行われる。また、その隣にあるのは［Interactive］ウィンドウを表示するためのボタンとなっている。

上に示した［概要］タブにはその環境についての概要が示されている。例えば、上の画面であれば、新規に作成する Python プロジェクトで使用される既定の環境が Python 3.6 になることが分かる。［対話型ウィンドウを開く］リンクはその名の通りの動作をする。［対話型のスクリプトを確認する］リンクをクリックすると、［Interactive］ウィンドウの起動時に自動的に実行されるスクリプトが保存されているディレクトリが Windows エクスプローラーに開かれる。［IPython 対話モードを使用する］チェックボックスをオンにすると、IPython を使用して、通常のものよりも高度な［Interactive］ウィンドウが表示されるようになる（ただし、IPython がインストールされている場合）。［PowerShell で開く］は現在のディレクトリで PowerShell を起動するものだ。その下には、環境がインストールされているパスと python コマンド、pythonw コマンドのパスが表示されている。

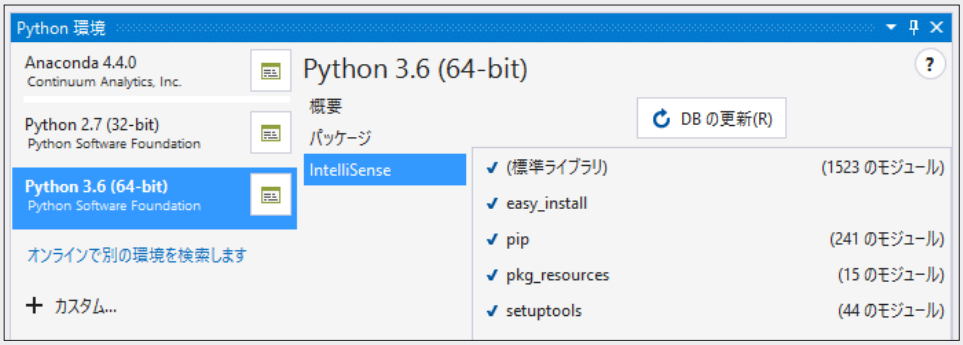
［パッケージ］タブを選ぶと、その環境にインストールされているパッケージが表示される。また、ここからインストール済みのパッケージを検索したり、PyPI（the Python Package Index）で配布されているパッケージを検索／インストールしたりできる。



［パッケージ］タブではパッケージの検索やインストールが可能

上の画面は Anaconda 4.4.0 環境にインストールされているパッケージを一覧したものだ。

最後の［IntelliSense］タブを選択すると、VS 2017 で IntelliSense を利用した入力で使用するデータベースの状態が表示される。



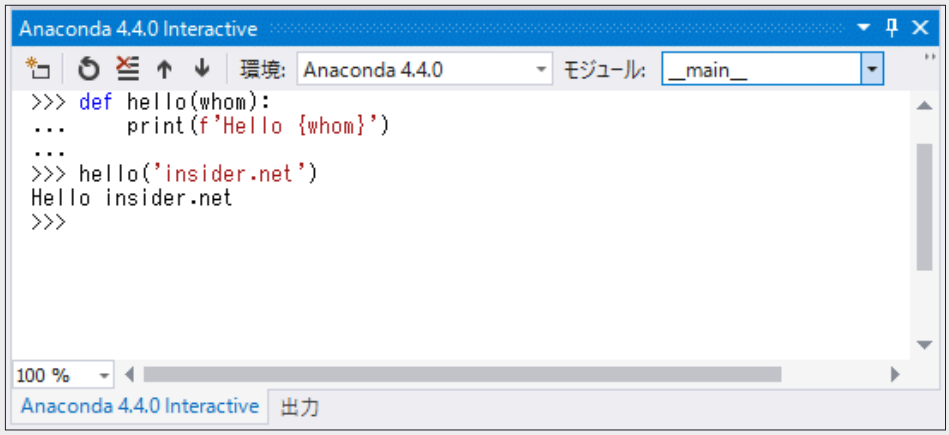
［IntelliSense］タブには IntelliSense が使用するデータベースが一覧される

→目次に戻る

[Interactive] ウィンドウ（対話型ウィンドウ）

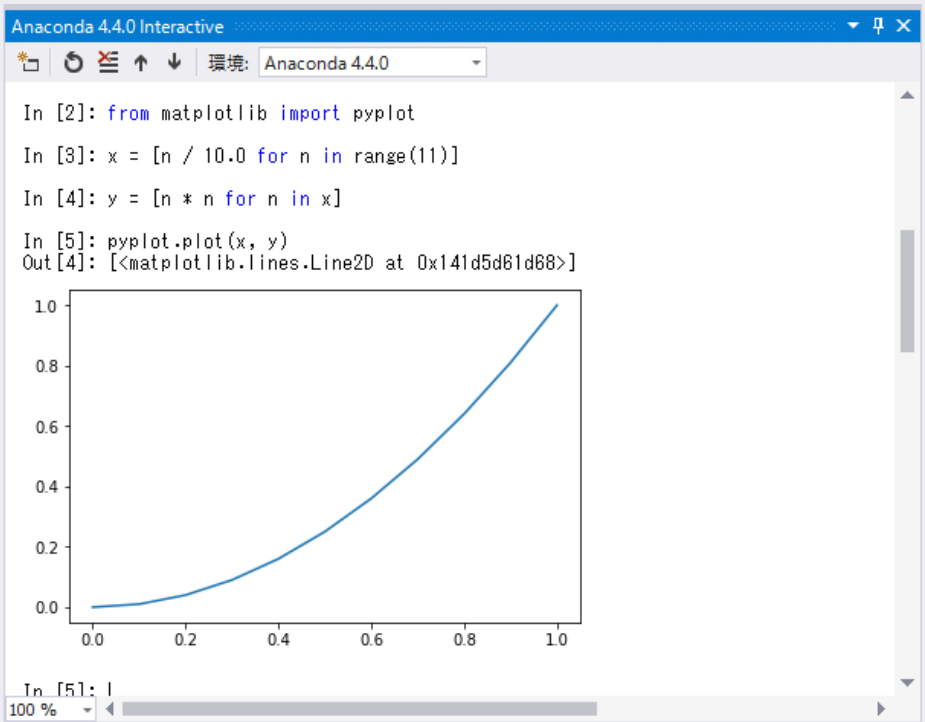
［概要］タブにある［対話型ウィンドウを開く］リンクをクリックすると、[Interactive] ウィンドウが表示される。このとき、IPython が利用可能な環境で［IPython 対話モードを使用する］チェックボックスをオンにしていれば、IPython ベースの［Interactive］ウィンドウとなる。

まずは通常の［Interactive］ウィンドウを起動してみよう。



通常の［Interactive］ウィンドウ

ここでは Anaconda 4.4.0 ベースの［Interactive］ウィンドウを表示して、関数 hello を定義し、それを実際に呼び出している。普通の REPL 環境だ。次に同じく Anaconda 4.4.0 環境で IPython を有効にした［Interactive］ウィンドウを表示してみよう。Anaconda 4.4.0 には標準で多数のモジュールが含まれているので、そこから matplotlib.pyplot モジュールを利用して、グラフをプロットしている。



IPython を利用した［Interactive］ウィンドウ

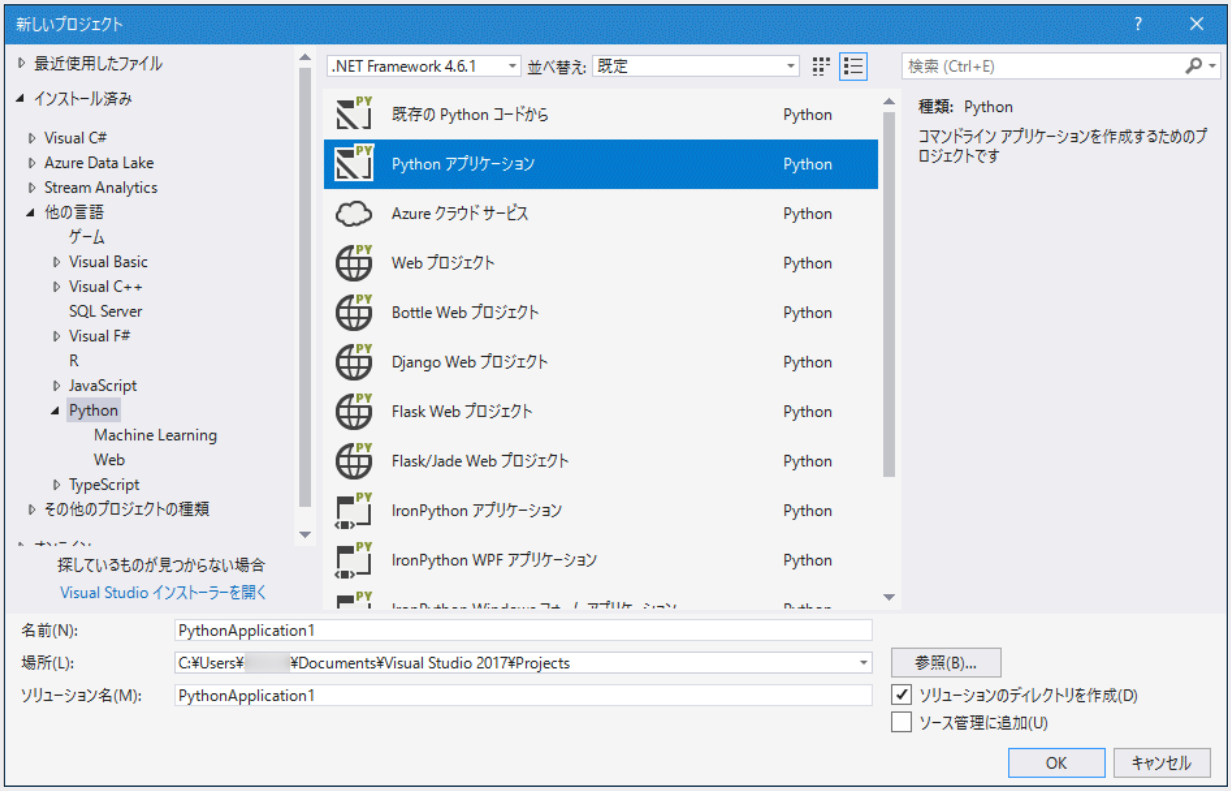
→目次に戻る

ここでは、X 軸の値として 0.0 ～ 1.0 の値を 0.1 刻みで、Y 軸の値としてそれを二乗した値を用いて、グラフをプロットしている。IPython を使うことで、通常の［Interactive］ウィンドウよりも高度な処理を対話的に行えることが分かるはずだ。

次に、Python のコンソールアプリを作成して、IDE の使い方を見てみよう。

Python プロジェクトの作成とソリューションエクスプローラー

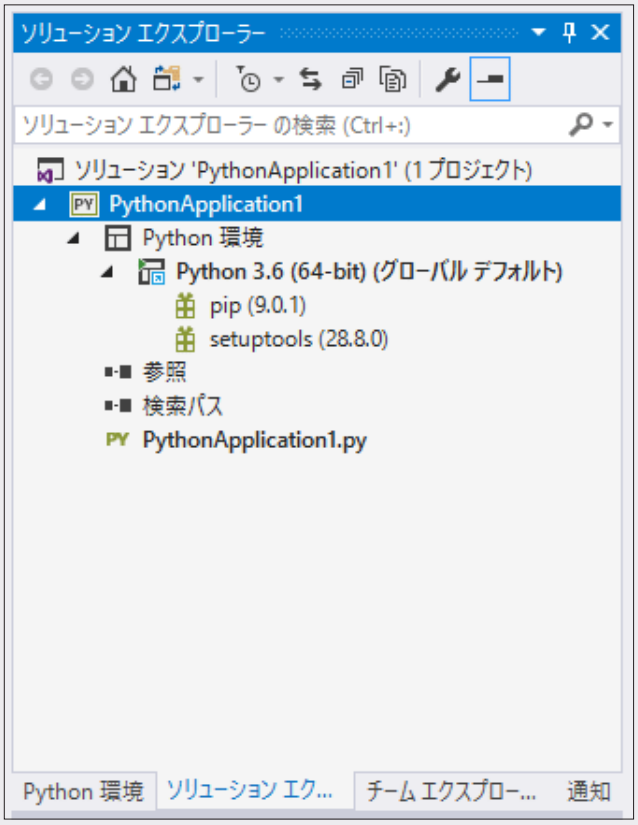
［Python 開発］ワークロード（と各種のサポート機能）をインストールすると、VS 2017 の［新しいプロジェクト］ダイアログで、Python 向けのプロジェクトテンプレートを選択できるようになる。



［新しいプロジェクト］ダイアログ

どのようなプロジェクトテンプレートが利用可能かについては「プロジェクト テンプレート」を参照されたい。ここでは、シンプルにコンソールアプリ用のプロジェクトテンプレートを使用して、簡単なアプリを作成している。ただし以下では、コードは書かずに、ソリューションエクスプローラーを中心にプロジェクトで使用する環境の構築について見ていく。

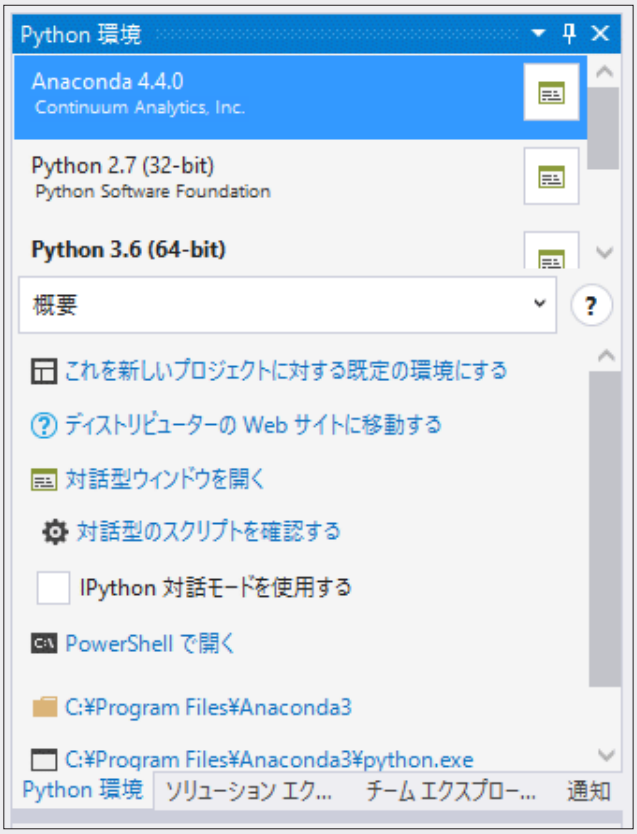
プロジェクト作成直後のソリューションエクスプローラーは次のようになっている。



コンソールアプリプロジェクトを作成した直後のソリューションエクスプローラー

［Python 環境］の下には、そのプロジェクトで利用可能な Python 環境が表示される。［参照］というのは、他のプロジェクトと同様に、このプロジェクトから他のプロジェクトや拡張機能への参照を追加したり、現在参照しているものを一覧したりするのに使用する。［検索パス］には Python が検索するパスや ZIP アーカイブを指定できる。

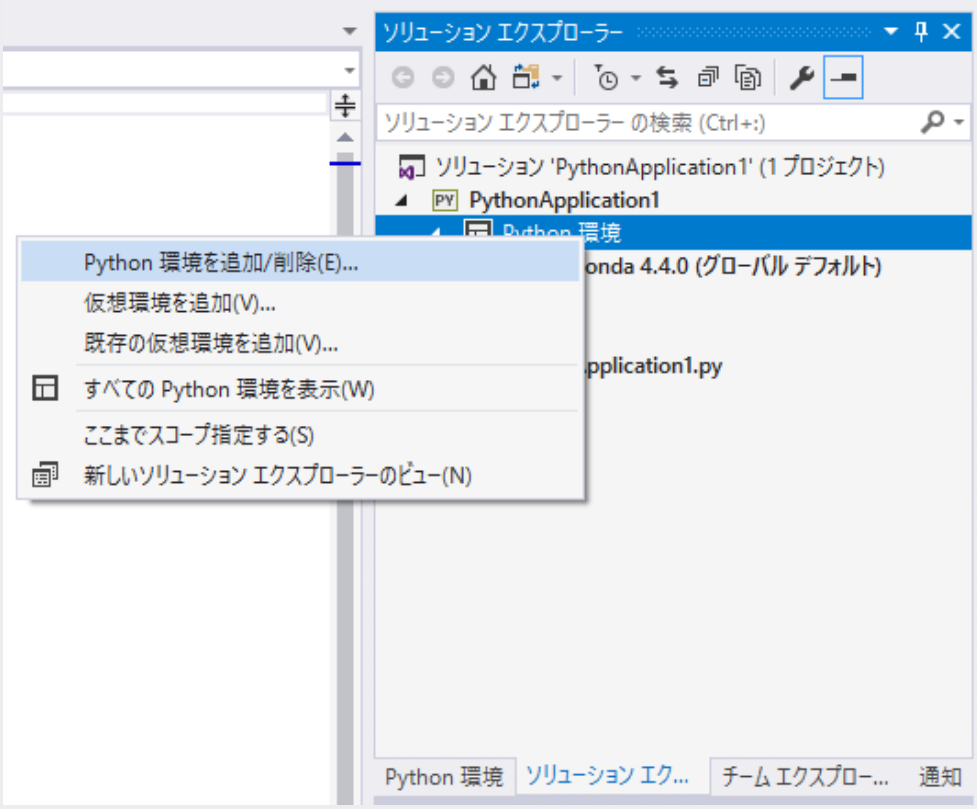
デフォルトでは、［Python 環境］の下には、［Python 環境］ウィンドウで「デフォルトの環境」として選択されているものが表示される。なお、デフォルトの環境を変更するには［Python 環境］ウィンドウの上部で、デフォルトの環境にしたいものを選び、［これを新しいプロジェクトに対する既定の環境とする］リンクをクリックする。



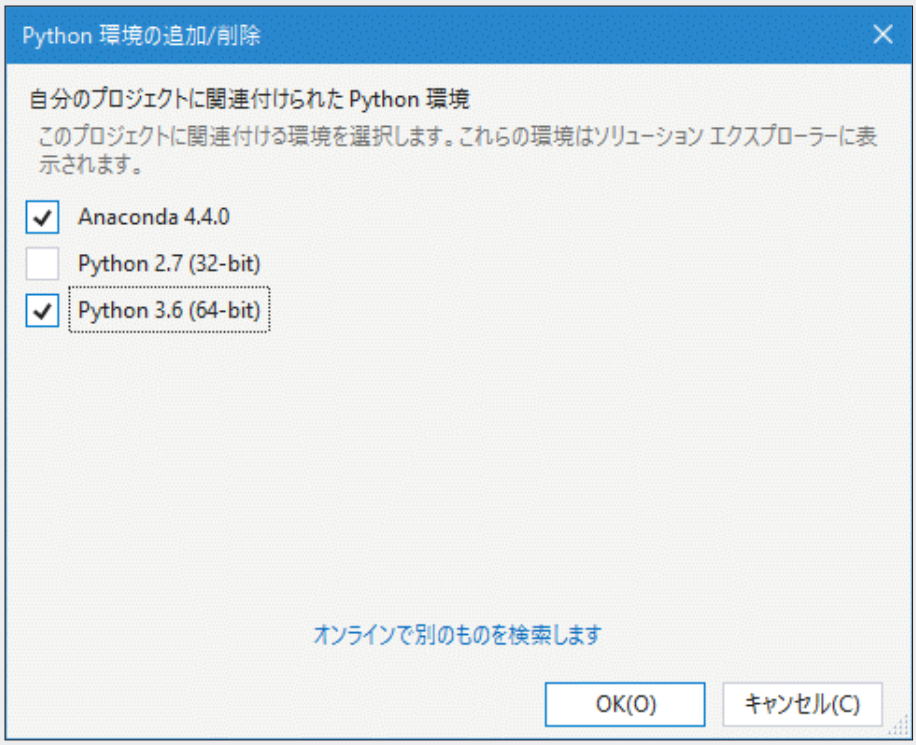
デフォルトの環境を変更する

上の画面では、Anaconda 4.4.0 が選択されている。ここで［これを新しいプロジェクトに対する既定の環境とする］リンクをクリックすれば、それ以降は Anaconda 4.4.0 がデフォルトの環境として使われるようになる。

使用する環境をプロジェクトごとに固定することも可能だ。これにはソリューションエクスプローラーで［Python 環境］を右クリックして、コンテキストメニューから［Python 環境を追加 / 削除］を選択する。すると、VS 2017 で利用可能な環境が表示されるので、そこからプロジェクトで使用する環境を選択すればよい。これにより、デフォルトに指定されていた Python 環境ではなく、指定した環境がそのプロジェクトで使用されるようになる。



［Python 環境を追加 / 削除］を選択

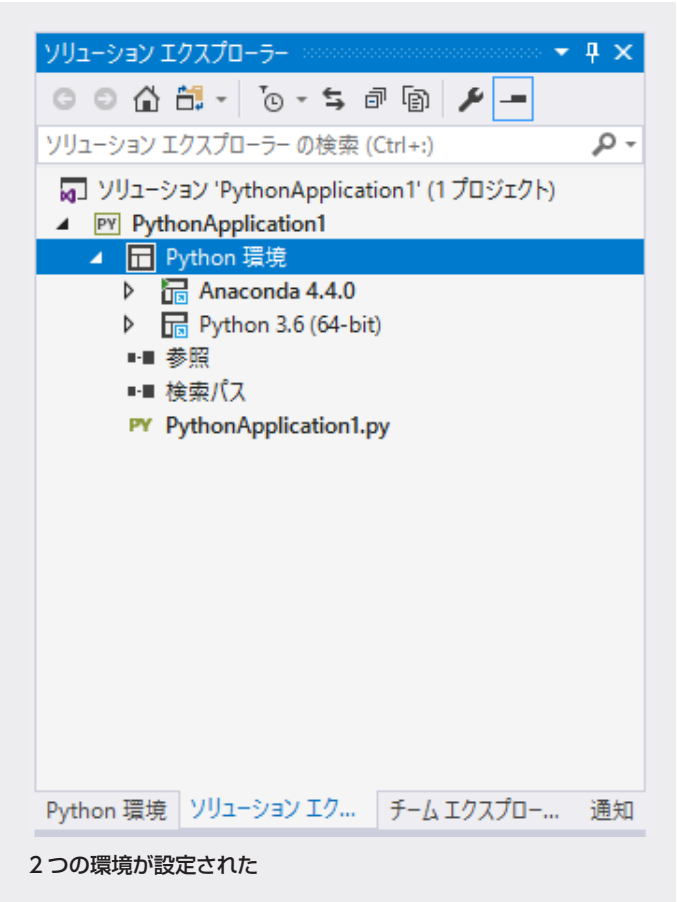


使用する環境を選択する

使用する環境の選択

→目次に戻る

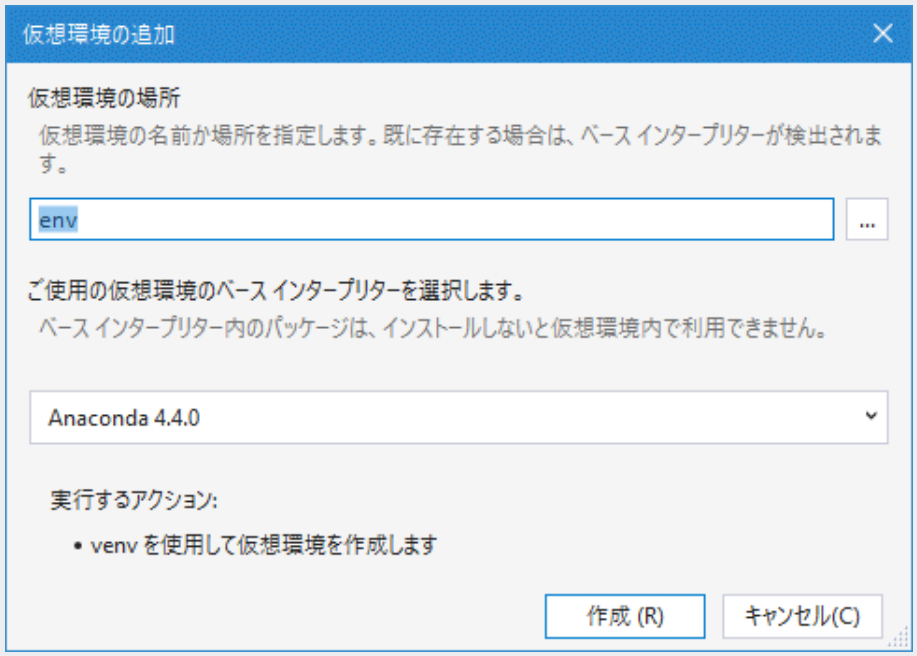
例えば、上のように Anaconda 4.4.0 と Python 3.6 を選択すると、ソリューションエクスプローラーの表示は次のようになる。



2 つの環境が設定された

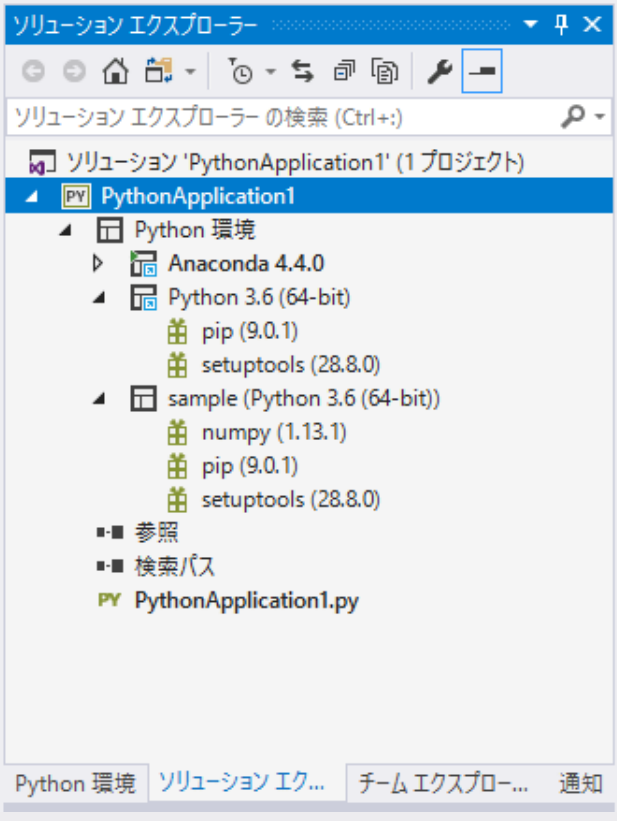
先ほど示したプロジェクト作成直後のソリューションエクスプローラーでは Python 3.6 の隣に「グローバル デフォルト」と表示されていたが、上の画像ではそういった表示はなくなっている。「グローバル デフォルト」というのは、それが全てのプロジェクトのデフォルト環境であることを示している。一方、「グローバル デフォルト」という文言がないのは、ここでは同じ環境（Python 3.6）を指定してはいるが、それは明示的にその環境を選択していることを意味している。なお、どの環境を使用するかを指定するには、使用したい環境を右クリックして、コンテキストメニューから［環境のアクティブ化］を選択する。

さらにプロジェクトごとに固有の「仮想環境」も構築できる。これは、特定の環境（Anaconda 4.4.0、Python 3.6 など）の上に、特定のパッケージなどをインストールした仮想的な環境だ。アプリで使用するパッケージや、そのバージョンなどはプロジェクトごとに異なるものになりがちだ。そうしたプロジェクト依存の情報をまとめて管理できるようにしたものが仮想環境である。VS 2017 で仮想環境を構築するには、ソリューションエクスプローラーで［Python 環境］を右クリックして、コンテキストメニューから［仮想環境を追加］を選択する。すると、次のようなダイアログが表示される。



［仮想環境の追加］ダイアログ

ここで仮想環境の名前と、ベースとなる環境を選択する。その後は、仮想環境に対して、パッケージのインストールなどを行う。例えば、Python 3.6 環境をベースとした「sample」という名前の仮想環境を作成して、そこに numpy パッケージを追加した状態のソリューションエクスプローラーを以下に示す。



仮想環境を追加した状態のソリューションエクスプローラー

→目次に戻る

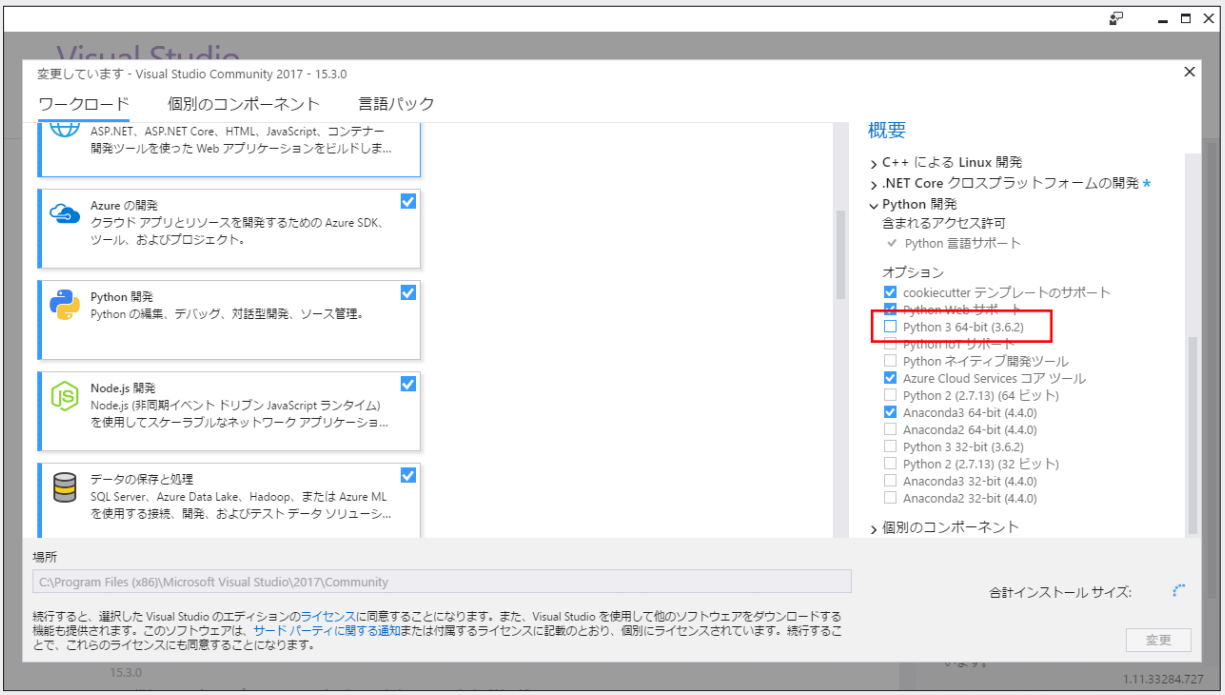
ベースとした Python 3.6 の環境には numpy パッケージがないが、sample の環境には numpy パッケージがインストールされていることが分かる。このようにベース環境に変更を加えることなく（あるいは、他のプロジェクトに影響を与えることなく）、プロジェクトに必要なパッケージの管理を個別に行えるのが仮想環境を構築する大きなメリットだ。なお、仮想環境を削除するには、削除したい仮想環境を右クリックして、コンテキストメニューから［削除］を選択する。

最後に［Python 開発］ワークロードを使用する上でのちょっとした注意点について触れておこう。

注意点

［Python 開発］ワークロードのインストール時に、事前にインストールしてある Python に対応するチェックボックスを［概要］ペーンや［個別のコンポーネント］タブでオンにすると、その後で［Python 開発］ワークロードをアンインストールするときにその処理系が同時にアンインストールされてしまうので気を付けよう。

例えば、既に Python 3.6.2（執筆時点で最新の処理系）をインストールしていて、その状態で［Python 開発］ワークロードをインストールしようとしたとする。このときに、このバージョンに対応する［Python 3 64-bit (3.6.2)］チェックボックスをオンにして、このワークロードをインストールすると、その後でこのワークロードをアンインストールしたときに、対応する Python 処理系まで同時にアンインストールされてしまう。これを避けるには、ワークロードのインストール時に［Python 3 64-bit (3.6.2)］チェックボックスをオフにしておく。



すでにインストール済みの Python 処理系については、対応するチェックボックスがあれば、それをオフにしておくのも 1 つの手（赤枠内）

上の画面では、すでに Python 3.6.2 がインストールしてあるがワークロードのアンインストール時に一緒にア

ンインストールされたくはないので、［Python 3 64-bit (3.6.2)］チェックボックスをオフにしている。

［Python 開発］ワークロードは、必要に応じて処理系のインストールからアンインストールまでをワークロードが一括して管理してくれるワンストップなツールであり、その管理下にインストール済みの処理系を含めるかどうかを、チェックボックスのオン／オフで決定していると考えるのがよいだろう。

最後にもう 1 つ。すでにローカル環境には存在していないバージョンの Python が［Python 環境］ウィンドウに表示されている場合には、レジストリの「HKEY_CURRENT_USER¥Software¥Python¥PythonCore」以下に余計なキーが残っていないかを確認してみよう。

◇ ◇ ◇ ◇ ◇ ◇

本章では［Python 開発］ワークロードのインストール、［Python 環境］ウィンドウ、［Interactive］ウィンドウ、ソリューションエクスプローラーでの Python 環境の取り扱いなど、VS 2017 における Python サポートについて見てきた。ここでは取り上げなかったが、［Python 開発］ワークロードでは IntelliSense を利用した快適なコード入力、デバッグなどの機能ももちろんサポートされている。さらに詳細を知りたいという場合には「Visual Studio での Python の使用」などのページを参照してほしい。

次章からは Python でプログラミングを行う上での基本となる構文や機能について見ていくことにしよう。

特集：Visual Studio で始める Python プログラミング

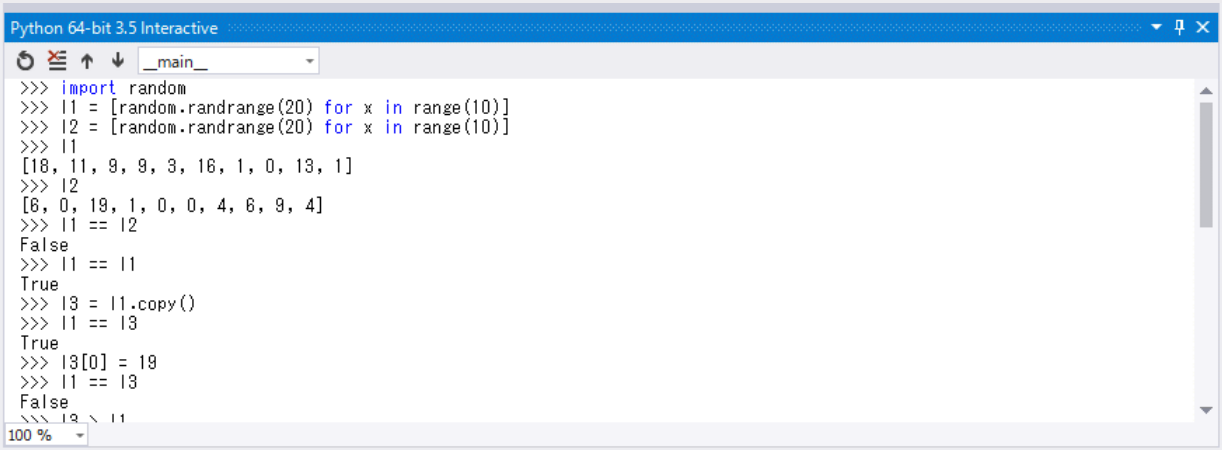
3. Python の文法、基礎の基礎

本章では、Python の制御構造と、リスト／タプル／辞書／集合という 4 つのデータ型について超速で見ていく。

ここでは、Python の制御構造と、リスト／タプル／辞書／集合という 4 つのデータ型について超速で見ていく。四則演算や基本データ型の種類、演算子の種類と優先順位など、通常の入門書にありそうな要素は省略する。また、C# などの言語の経験者を対象として「ふ〜ん、Python ではそうなっているのね」と感じてもらうことを目的として、概要をさらりと眺めるだけで詳細には触れないので、詳しく知りたい方は Python のドキュメントや各種の入門書を参照してほしい。

変数定義と制御構造

以下では PTVS（あるいは［Python 開発］ワークロード）の［Interactive］ウィンドウを使って、変数定義、制御構造といった基本的な構文要素を見ていこう。ここでは Anaconda 4.1.1 で提供される Python 3.5.2 を使用している。



［Interactive］ウィンドウ

ここでは［Interactive］ウィンドウとの対話だけで話を進めていく。プロジェクトは作成しない。まずは以下からだ。

```
>>> a = 100
>>> a
100
>>> b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
>>> type(a)
<class 'int'>
>>> a = 'insider.net'
>>> type(a)
<class 'str'>
```

変数定義

上記の結果からまず分かるのは、代入をすることで変数が定義されることだ（変数宣言は不要。未定義の変数を使おうとすると例外「TypeError」が発生する）。変数に型はない。型を持つのは変数が参照しているオブジェクトであり、ある変数に異なる型のオブジェクトを代入することも可能だ（それが推奨されるかどうかは別の話）。なお、単純なオブジェクトの型は type 関数で調べられる（継承関係を調べるには isinstance 関数の利用が推奨される）。

次に Python の制御構造を幾つか紹介する。以下は 1 ～ 10 を要素とする配列（Python では実際にはこれはリスト）の中から偶数を表示するコードだ。変数 a が参照しているのはリストオブジェクトだ（後述）。

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> i = 0
>>> while i < len(a):
...   if not a[i] % 2:
...     print(a[i])
...   i += 1
...
2
4
6
8
10
```

while 文と if 文

見ての通り、while 文も if 文も基本的な構文は C# などのそれとそう変わらない。ただし、C 系統の言語とは異なり、条件式を囲むかっこは不要だ。また、while 文にしても if 文にしても、そのボディー（コードブロック）はインデントを付けて表現する。この場合は if 文と次の代入文が while 文のコードブロックとなる。if 文のコードブロックは print 関数呼び出しとなる。

なお、len 関数は引数に渡したオブジェクトの長さ／要素数を返す、Python の組み込み関数である。もう 1 つ。Python にはインクリメント／デクリメント演算子はないので、変数の値を 1 加算するには上記のように加算代入演算子を使用する。

それから、if 文の条件式を見ると分かる通り、Python ではブーリアン値以外の値も条件として評価される。

while 文の構文を以下に示す。else 節に記述した内容は条件式が成立しなくなった時点で実行される（ループ中で break 文が実行された場合には、else 節は実行されないので、これを利用してある条件に応じて、while ループ終了後の処理に変化を付けられる。ただし、利用頻度は高くはないだろう）。

```
while 条件式:
  ボディー
else:
  ボディー
```

while 文の構文

if 文の構文は次の通りだ。C# などの言語と明確に異なるのは、「else if」ではなく「elif」キーワードを使用する点だ。もちろん、elif 節と else 節は省略可能である。

```
if 条件式1:
  ボディー
elif 条件式2:
  ボディー
else:
  ボディー
```

if 文の構文

なお、Python には switch 文は存在しない。条件分岐を行う場合には if 文を使用する。for 文はもちろん存在している。以下にその利用例を示す。これは上の while ループを for 文で書き直したものだ。

```
>>> for item in a:
...   if not item % 2:
...     print(item)
```

for 文の利用例

これは C 系統の言語に見られる「for (初期化式 ; 継続条件式 ; ループ式)」形式の for 文とは異なる。むしろ、C# の foreach 文に近い形式のものだ。for 文の構文を以下に示す。

```
for ループ内で使用する変数 in 式:
  ボディー
else:
  ボディー
```

for 文の構文

上の「式」は反復可能なオブジェクトを返すものを記述する。反復可能なオブジェクトとは、そのオブジェクトが含む要素を一度に 1 つ返送可能なオブジェクトのことだ。例えば、上では Python のリストを渡している。これ以外には、文字列、タプルなどの「シーケンス型」（後述）に分類されるオブジェクトや、辞書やファイル、__iter__ ／__getitem__ メソッドを持つクラスのインスタンスなどが反復可能なオブジェクトとして扱える（詳細については「用語集」などからリンクをたどってほしい）。

反復可能なオブジェクトを返す典型的な式としてはもう 1 つ、range 関数がある。この関数の戻り値である range オブジェクトは、リストのように各要素をメモリ上に確保するのではなく、指定された範囲／差分を管理するオブジェクトであり、反復処理を行うたびにそれが管理している範囲／差分を基に整数列を生成してくれる。以下に例を示す。

```
>>> range(1, 11, 1)
range(1, 11)
>>> for num in range(1, 11, 1):
...   if not num % 2:
...     print(num)
...
```

range 関数の使用例

range 関数の引数は順に「初期値、終端値、差分」となっている（終端値は生成される数列には含まれないので、上では「11」を指定している。つまり、これで範囲 1 ～ 10、差分 1 の等差数列が生成される）。あるいは終端値のみを指定する「range(終端値)」という呼び出しも可能だ（この場合、初期値は 0、差分は 1 となる）。

リストとタプル

基本データを組み合わせて、より複雑な構造のデータを作成するには以下のものが使える。

- リスト：任意の型のデータを要素とするデータ構造。[] で囲んで記述する。変更可能
- タプル：任意の型のデータを要素とするデータ構造。() で囲んで記述する。変更不可能
- 辞書：いわゆる連想配列。{} で囲んで記述する。変更可能

- 集合: 順序を持たない要素のコレクション。集合には 1 つの要素を重複して格納することはできない。変更可能な集合と、変更不可能な frozenset がある
- クラス : 属性とメソッドで構成されるユーザー定義型
- モジュール : 関連ある関数や変数、クラスなどを含む

これらのうち、リスト、タプルは「シーケンス型」に分類される。シーケンス型は順序を持った要素の集合を表す（実際には、シーケンス型に分類されるデータ型としては文字列、bytes 型、bytearray 型なども存在する）。

これらのうち、以下ではリスト、タプル、辞書、集合について見てみよう。

リスト

リストは、さまざまな型のデータを要素とし、それらを [] の中にカンマで区切って並べていく。実体は list クラスのインスタンスとなる。以下に例を示す。

```
>>> lst = [5, 7, 2, 9, 3] # リストの作成
>>> type(lst)  # リストはlistクラスのインスタンス
<class 'list'>
>>> lst[4]     # インデックス指定
3
>>> lst[0:4]   # リストのスライス
[5, 7, 2, 9]
>>> lst[0:4:2] # リストのスライス（インデックス0～3の要素を1つ飛ばしで）
[5, 2]
>>> lst[-1]    # インデックスに負数を指定すると末尾からアクセス可能
3
>>> lst[0:2] = [4, 6]   # スライスされた範囲の書き換え
>>> lst
[4, 6, 2, 9, 3]
>>> lst[0:2] = [5, 7, 1] # この例では左辺と右辺の要素数が違っていてもOK
>>> lst
[5, 7, 1, 2, 9, 3]
>>> sorted(lst)    # sorted関数はリストをソートしたもののコピーを返す
[1, 2, 3, 5, 7, 9]
>>> lst          # 元のリストは変化していない
[5, 7, 1, 2, 9, 3]
>>> lst.sort()     # listクラスのsortメソッドはリストを並べ替える（破壊的変更）
>>> lst
[1, 2, 3, 5, 7, 9]
```

リストの使用例

上のコードで定義しているリストでは、要素は全て整数値となっているが、複数の型のオブジェクトを要素にまとめることも可能だ。リストの要素は「lst[4]」のようにインデックスを指定して取り出したり、その値を変更したりできる。

「lst[0:4]」は「スライス」表記と呼び、リスト内の指定した要素を取り出すものだ（スライスは指定された範囲の浅いコピーを返送する）。このときには、[] 内に「開始インデックス、終了インデックス、差分」を順に記述していく。ただし、終了インデックスで指定される要素はスライス操作の対象にはならないことに注意（終了インデックスで指定される要素の直前の要素までがスライスに含まれる）。代入文の左辺に置けば、それらを置き換えることもできる（差分が 1 以外の場合は、スライスと代入するリストの要素数が等しくなければならない）。

リストは変更可能なオブジェクトであり、以下のような操作が行える。

- 要素の追加：append ／ extend メソッドによるリスト末尾への要素の追加
- 要素の挿入：insert メソッドによるリスト中の指定した箇所への要素の挿入
- 要素の削除：remove メソッドで指定した値を持つ最初（インデックスが最小）の要素を削除
- 要素のポップ: pop メソッドによる指定したインデックス位置の要素の取得とリストからの削除（インデックスを指定しなければ、末尾の要素が対象となる。このため、append ／ pop メソッドを使うことで、リストをスタックとして扱える）
- 全要素の削除：clear メソッドによるリストの全要素の削除
- 要素の検索：index メソッドで指定した値を持つ最初（インデックスが最小）の要素を取得
- 要素の数え上げ：count メソッドで指定した値がリスト中に何個あるかを計算
- リストの並べ替え: sort メソッドによるリストのソート。上で見た通り、リスト自身を書き換える。書き換えをせずに、並べ替えた結果のコピーを取得するには組み込みの sorted 関数を使用する
- リストの逆順化：reverse メソッドによるリストの要素の逆順化（破壊的操作）
- リストのコピー：copy メソッドまたはスライスによるリストの浅いコピーの取得

以下に例を幾つか示す。

```
>>> lst = [1, 2, 3]
>>> lst.append(4)  # appendメソッドによる要素追加
>>> lst
[1, 2, 3, 4]
>>> lst.extend([5, 6])  # extendメソッドによる要素追加
>>> lst
[1, 2, 3, 4, 5, 6]
>>> lst.extend((7,8))  # extendメソッドにはタプルなど反復可能なオブジェクトを渡せる
>>> lst
[1, 2, 3, 4, 5, 6, 7, 8]
>>> lst += [9]          # 単一要素からなるリストの加算代入
>>> lst
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> lst += 10           # リストに整数を加算と解釈されるのでエラーになる
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
>>> lst.remove(6)       # リスト中の最初の「6」を削除
>>> lst
[1, 2, 3, 4, 5, 7, 8, 9]
>>> lst.insert(5, 6)    # インデックス5の位置に要素「6」を挿入
>>> lst
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> lst.index(5)        # 要素「5」のインデックスを取得
4
>>> lst.append(1)
>>> lst
[1, 2, 3, 4, 5, 6, 7, 8, 9, 1]
>>> lst.count(1)        # 要素「1」の出現回数を取得
2
>>> lst.pop()           # 末尾の要素をポップ
1
>>> lst
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> lst.reverse()       # リストを逆順に並べ替える
>>> lst
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

リストの操作

ここでは示さなかったが、特定要素またはリスト全体の削除には del 文も利用できる。

タプル

タプルはリストと同様だが、[] ではなく、() の中にその要素をカンマで区切って並べていく（実際には () はなくても構わない）。以下に例を示す。

```
>>> tpl = (5, 7, 2, 9, 3)  # タプルの作成
>>> type(tpl)  # タプルはtupleクラスのインスタンス
<class 'tuple'>
>>> tpl = 5, 7, 2, 9, 3    # かっこがなくてもよい
>>> tpl
(5, 7, 2, 9, 3)
>>> tpl[4]      # インデックスによるアクセス
3
>>> tpl[0:4]    # タプルのスライス
(5, 7, 2, 9)
>>> tpl[0:4:2]  # タプルのスライス（インデックス0～3の要素を1つ飛ばしで）
(5, 2)
>>> tpl[-1]     # インデックスに負数を指定
3
>>> tpl[0:2] = (4, 6)  # タプルは変更不可能
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> sorted(tpl) # sorted関数によるタプルの要素の並べ替え
[2, 3, 5, 7, 9]
>>> type((1))  # 要素を1つだけかっこで囲んだ場合にはタプルにはならない
<class 'int'>
>>> type((1,))  # その場合は末尾にカンマを付加する
<class 'tuple'>
```

タプルの使用例

リストとタプルの大きな違いは前者が変更可能（ミュータブル）であるのに対して、後者は変更不可能（イミュータブル）であることだ。そのため、上の例で示したように、タプルではその要素への代入はサポートされていない。ただし、次のようにタプルの要素にリストを含めることは可能だ。この場合、リストは変更可能なので、その要素を変更できる。

```
>>> lst2 = [1, 2, 3]
>>> lst3 = [4, 5, 6]
>>> tpl2 = (lst2, lst3)
>>> tpl2[0][1] = 4  # タプルの要素であるリストの要素を変更
>>> tpl2
([1, 4, 3], [4, 5, 6])
>>> tpl[0] = [7, 8, 9]  # タプル自体は変更不可能なので、その要素に代入はできない
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

リストを要素とするタプル

また、上で見たようなリストでサポートされている破壊的操作（メソッド）もタプルには提供されていない。例えば、リストの例で見た sort メソッドはタプルにはない。タプル（tuple クラス）でサポートされているメソッドは、指定した値の出現回数を数え上げる count メソッドと指定した値を持つ最初の要素のインデックスを取得する index メソッドのみである。

辞書

辞書は「キー／値」の組の集合である。連想配列といってもよい。リストやタプルは順序を持つ（それらの大小を比較できる）が、辞書には順序がない。以下に例を示す。

```
>>> d = { 'forum1': 'Windows Server Insider',  # 文字列をキーとした辞書の作成
...       'forum2': 'Insider.Net',
...       'forum3': 'Server & Storage' }
...
>>> d['forum1']
'Windows Server Insider'
>>> d.items()  # キー／値の列挙
dict_items([('forum1', 'Windows Server Insider'), ('forum3', 'Server & Storage'),
('forum2', 'Insider.Net')])
>>> d.keys()   # キーの列挙
dict_keys(['forum1', 'forum3', 'forum2'])
>>> d.values() # 値の列挙
dict_values(['Windows Server Insider', 'Server & Storage', 'Insider.Net'])
>>> d['forum4'] = 'Coding Edge'  # 要素の追加
>>> d
{'forum1': 'Windows Server Insider', 'forum3': 'Server & Storage', 'forum2': 'Insider.Net',
'forum4': 'Coding Edge'}
>>> d.update({'forum3': 'System Insider'})  # 要素の書き換え
>>> d.update(forum4='Server & Storage')     # 要素の書き換え
>>> d
{'forum1': 'Windows Server Insider', 'forum3': 'System Insider', 'forum2': 'Insider.Net',
'forum4': 'Server & Storage'}
>>> d2 = { 0: 'Windows Server Insider',  # 整数値をキーとした辞書の作成
...       1: 'Insider.NET',
...       2: 'Server & Storage' }
...
>>> d2[0]
'Windows Server Insider'
>>> d3 = { (0, 0): 'Windows Server Insider',  # タプルをキーとした辞書の作成
...       (0, 1): 'Insider.NET',
...       (1, 0): 'Server & Storage' }
...
>>> d3[(1,0)]
'Server & Storage'
>>> d4 = dict([('forum1', 'Windows Server Insider'), # dict()関数を使用
...           ('forum2', 'Insider.NET'),
...           ('forum3', 'Server & Storage')])
...
>>> d4
{'forum1': 'Windows Server Insider', 'forum3': 'Server & Storage', 'forum2':
'Insider.NET'}
>>> d5 = dict(forum1='Windows Server Insider', forum2='Insider.NET')
>>> d5
{'forum1': 'Windows Server Insider', 'forum2': 'Insider.NET'}
```

辞書の使用例

→目次に戻る

辞書オブジェクトは dict クラスのインスタンスであり、上で見たように、キー／値の組、キー、値のそれぞれについて辞書ビュー（辞書の内容の変更に動的に追随する、反復可能なオブジェクト）を返送する items メソッド、keys メソッド、values メソッドを持つ。

また、組み込みの dict 関数を使用すると、2 つの要素で構成されるタプル（を含むリストなど）から辞書を作成できる。最後の例では、dict 関数にキーワード引数とその値を指定して辞書を作成している。

Python の辞書で重要なのは、キーには不変値を含むことはできないこと。簡単にいえば、リストや辞書を（別の）辞書のキーとすることはできないことに注意（リストを要素に含むタプルもキーにはできない）。

集合

集合はその名の通り、数学的な意味での集合を表すためのものだ。集合にある値が含まれているかどうかのチェックや、集合間の各種演算がサポートされている。辞書と同様に、集合にも順序はない。また、集合には変更可能な「集合」（set）と変更不可能な「frozenset」の 2 種類がある。

以下に例を示す。集合は辞書と同様に {} を使用して記述する（ただし、辞書はキー／値の組を並べていったが、集合では単一要素を並べていく）。また、frozenset は作成したら変更はできないので、何らかの要素を基にして frozenset 関数を呼び出すことで作成する。

```
>>> s1 = { "Windows Server Insider", "Insider.NET", "System Insider" }
>>> s2 = s1.copy()
>>> s2.add("Coding Edge")
>>> s1
{'System Insider', 'Windows Server Insider', 'Insider.NET'}
>>> s2
{'System Insider', 'Windows Server Insider', 'Coding Edge', 'Insider.NET'}
>>> s2 - s1  # 差集合
{'Coding Edge'}
>>> s2.difference(s1)    # 差集合
{'Coding Edge'}
>>> s2.intersection(s1)  # 共通集合
{'System Insider', 'Windows Server Insider', 'Insider.NET'}
>>> s1 >= s2  # s1がs2を包含しているか
False
>>> s2 >= s1  # s2がs1を包含しているか
True
>>> s1 ^ s2  # 対称差（どちらかにのみ含まれている要素からなる集合）
{'Coding Edge'}
>>> s1.add("Server & Storage")  # s1に要素を追加
>>> s1 ^ s2  # 対称差
{'Server & Storage', 'Coding Edge'}
>>> s2 > s1  # もはやs1はs2の部分集合ではない
False
>>> s3 = { "Build Insider" }
>>> s3.isdisjoint(s1)  # s3とs1は互いに素（共通する要素を持たない）
True
>>> fs = frozenset(s1) # s1を基にfrozensetを作成
>>> fs
frozenset({'Server & Storage', 'System Insider', 'Windows Server Insider', 'Insider.NET'})
>>> fs.add("Build Insider")  # frozensetは変更不可能
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

集合の利用例

set 関数を使うと、リストなど、他のオブジェクトから集合を作成することもできる。

```
>>> lst
[9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> s4 = set(lst)  # set関数を利用してリストから集合を作成
>>> s4
{1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> d
{'forum1': 'Windows Server Insider', 'forum3': 'Server & Storage', 'forum2': 'Insider.Net'}
>>> s5 = set(d)  # set関数を利用して辞書から集合を作成。値は落とされ、キーだけが残る
>>> s5
{'forum1', 'forum3', 'forum2'}
>>> s6 = set(d.items())  # set関数に辞書ビュー（反復可能オブジェクト）を渡して、集合を
作成
>>> s6
{('forum2', 'Insider.Net'), ('forum3', 'Server & Storage'), ('forum1', 'Windows Server
Insider')}
```

set 関数の使用例

→目次に戻る

コンテナに共通の操作

ここまでに見てきたリスト、タプル、辞書、集合などの「コンテナ」オブジェクト（他の要素への参照を格納するオブジェクト）には共通する操作があるので、ここで幾つか見ておこう。最初にコレクションを適当に作成しておく。

```
>>> l = list(range(10))  # リスト、タプル、辞書、集合の作成
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> t = tuple(zip(l, "abcdefghij"))
>>> t
((0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e'), (5, 'f'), (6, 'g'), (7, 'h'), (8, 'i'), (9, 'j'))
>>> d = dict(t)
>>> d
{0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 5: 'f', 6: 'g', 7: 'h', 8: 'i', 9: 'j'}
>>> s = set(l)
>>> s
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

コンテナオブジェクトの作成

上ではリストを基に、適当に他の 3 種類のコレクションを作成した。タプルの作成で使用している zip 関数は引数に渡されたシーケンス型のオブジェクトまたは反復可能なオブジェクトを基に、それらを組にしたタプル（のイテレータ）を返送するものだ。ここではそれを tuple 関数に渡して整数と文字列を要素とするタプルを作成している。

まずここで見たコンテナは全て反復可能（iterable、イテラブル）である。反復可能なオブジェクトは、一度に 1 つずつその要素を返送できる。これらは上で見たような for ループでの反復処理に使用できる。以下に例を示す（実行結果は省略）。

```
>>> for num in l:
...     print(num)
>>> for num, c in t:
...     print("{0}: {1}".format(num, c))
>>> for num, c in d.items():
...     print("{0}: {1}".format(num, c))
>>> for num in s:
...     print(num)
```

コンテナの反復処理

また、コンテナに要素が含まれているかどうかの存在判定も可能である。これは「値 in コンテナ」もしくは「値 not in コンテナ」の形式で判定できる。

```
>>> import random # randomモジュールのインポート
>>> l1 = [random.randrange(20) for x in range(10)]  # リストの内包表記による乱数リストの作成
>>> l2 = [random.randrange(20) for x in range(10)]
>>> l1
[3, 10, 16, 8, 0, 18, 14, 15, 12, 11]
>>> l2
[8, 12, 6, 13, 8, 19, 17, 0, 4, 5]
>>> l1 == l2  # リストの比較
False
>>> l1 > l2
False
>>> l2 > l1
True
>>> l3 = [8, 12, 6, 13, 8, 19, 17, 0, 4, 6] # 最終要素だけl2と違うリストl3を作成
>>> l3 > l2   # l3の方がl2よりも大きい
True
>>> t1 = tuple(l1)  # l1を基にタプルt1を作成
>>> t1
(3, 10, 16, 8, 0, 18, 14, 15, 12, 11)
>>> max(t1)   # t1に含まれる要素の最大値
18
>>> min(t1)   # t2に含まれる要素の最小値
0
```

存在判定

コンテナに含まれる要素の数は組み込みの len 関数で取得できる。

リストとタプルで共通の操作

リストとタプルの 2 つのシーケンス型ではさらに共通に利用できる操作が提供されている。例えば、先ほど見た要素のスライスがそうだ。また、index メソッドによる要素の検索、count メソッドによる出現回数の数え上げもそうである。これ以外には、リストやタプルの比較、連結、最小値／最大値の取得などが挙げられる。以下に例を示す。

```
>>> import random # randomモジュールのインポート
>>> l1 = [random.randrange(20) for x in range(10)]  # リストの内包表記による乱数リストの作成
>>> l2 = [random.randrange(20) for x in range(10)]
>>> l1
[3, 10, 16, 8, 0, 18, 14, 15, 12, 11]
>>> l2
[8, 12, 6, 13, 8, 19, 17, 0, 4, 5]
>>> l1 == l2  # リストの比較
False
>>> l1 > l2
False
>>> l2 > l1
True
>>> l3 = [8, 12, 6, 13, 8, 19, 17, 0, 4, 6] # 最終要素だけl2と違うリストl3を作成
>>> l3 > l2  # l3の方がl2よりも大きい
True
>>> t1 = tuple(l1)  # l1を基にタプルt1を作成
>>> t1
(3, 10, 16, 8, 0, 18, 14, 15, 12, 11)
>>> max(t1)  # t1に含まれる要素の最大値
18
>>> min(t1)  # t2に含まれる要素の最小値
0
```

リストとタプルで共通の操作

→目次に戻る

文字列

本項の最後に文字列操作（の一部）についても紹介しておこう。というのは、Python では文字列もシーケンス型に分類されるからだ。つまり、文字列は文字が連続したもので、順序性を持ち、反復可能である。そのため、その要素のスライスも可能だし、辞書式順序での比較が可能だ。

```
>>> s = "Insider.net"
>>> s
'Insider.net'
>>> for c in s:  # 反復処理
...   print(c)
...
I
n
s
i
d
e
r
.
n
e
t
>>> s[-4:]  # 文字列のスライス（インデックス-4から文字列末尾までをスライス）
'.net'
>>> s1 = s.upper()
>>> s2 = s.lower()
>>> s1
'INSIDER.NET'
>>> s2
'insider.net'
>>> s1 < s2  # 文字列を辞書式順序で比較
True
```

文字列をシーケンスとして扱う

◇ ◇ ◇ ◇ ◇ ◇

本章では Python の構文の基礎知識と、リストをはじめとする各種のコンテナオブジェクトについて見てきた。次章では関数やクラス、リストの高度な話題について紹介する。

特集：Visual Studio で始める Python プログラミング

4. Python の関数、超速入門

本項は Python の関数――通常の名前付き関数、ラムダ式で作成する無名関数、反復的に値を返すジェネレーター ――について見ていこう。

前章では、Python の基本構文を幾つかと、リスト／タプル／辞書／集合という 4 つのデータ構造を紹介した。これらは Python でコードを書く上で必須となる要素だ。本章ではもう 1 つの大きな要素である関数を取り上げる。これまでと同様、既に他の言語の経験がある方を対象に、例を交えながら超速で関数を見ていこう。

関数

Python で関数を定義するには def キーワードを使用する方法と lambda キーワードを使用する方法がある。前者は名前付きの関数を、後者はラムダ式（無名関数）を定義するものだ。まずは通常の（名前付き）関数の定義から見てみよう。ここでも VS の［Interactive］ウィンドウを利用して、対話的に Python コードを入力していく。

def キーワードによる関数定義

以下に def キーワードによる関数定義の基本構文を示す。

```
def 関数名(パラメーターリスト):
  関数定義のボディー
```

関数定義文の構文

関数定義文は実行可能な文であり、Python 処理系がこれを実行することで、その名前が識別子として導入され、その識別子と関数オブジェクトが関連付けられる。

簡単な例を以下に示す。

```
>>> def hello(to):
...   print("hello {0}".format(to))
...
>>> hello("insider.net")
hello insider.net
>>> greet = hello
>>> greet("insider.net")
hello insider.net
```

関数 hello の定義と呼び出し

→目次に戻る

これはパラメーターを 1 つ取り、その値と文字列を書式化したものをコンソールに出力する関数だ。Python では関数もオブジェクトであるため、変数 greet に関数 hello を代入して、その後、「greet(...)」のように呼び出すことも可能だ。

関数の引数

関数のパラメーター名は、その呼び出し時に「キーワード引数」のキーワードとして利用できる。また、パラメーターにはデフォルト値（デフォルト引数）を指定することも可能だ。以下に例を示す。

```
>>> def calc(a, b, op='+'):
...   if op not in "+-*/":
...     return
...   print(eval("{0} {1} {2}".format(a, op, b)))
...
>>> calc(1,1)
2
>>> calc(1, 2, '-')
-1
>>> calc(1, op='*', b=0.5)
0.5
```

キーワード引数とデフォルト引数

関数 calc は 3 つのパラメーターを取る。a と b の 2 つのパラメーターには数値を、パラメーター op には四則演算を表す記号のいずれかを指定してこの関数を呼び出すとその結果がコンソールに出力される。パラメーター op にはデフォルト値「'+'」が指定されている（他の 2 つにはデフォルト値はない）。

最初の呼び出し例では第 3 引数を省略している（デフォルト引数を利用）。次の呼び出し例では 3 つの引数を全て指定している。これらの引数は「位置指定引数」（positional argument）と呼ばれ、引数リストに並べた順番に引数がパラメーターに引き渡される。最後の呼び出し例では、第 1 引数は位置指定、残る 2 つについては「キーワード引数」形式で引数がパラメーターに渡されている。

キーワード引数と位置指定引数を混在させて関数を呼び出す場合には、キーワード引数よりも先に位置指定引数を指定する必要がある。また、デフォルト値を持たないパラメーターについては呼び出し時に引数を指定する必要もある。以下に例を示す。

```
>>> calc(op='*', 2, 3)  # 位置指定引数をキーワード引数より後に指定
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
>>> calc(1, op='-')     # デフォルト値を持つのは第3引数のみ
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: calc() missing 1 required positional argument: 'b'
```

誤った関数呼び出し

→目次に戻る

可変数個の引数については、Python ではタプルもしくは辞書を利用して受け取る。一般的な引数はタプルに、キーワード引数は辞書に受け取る形になる。以下に例を示す。まずはタプルを利用する場合だ。

```
>>> def foo(a, b, *args):  # 3つ目からはタプルargsに受け取る
...   print("contents of args")
...   for arg in args:
...     print(arg)
...
>>> foo(1, 2, 100, ["insider.net", "windows server insider"])
contents of args
100
['insider.net', 'windows server insider']
```

タプルで可変数個の引数を受け取る

タプルで受け取る場合には、そのパラメーター名の前に「*」を前置する。このとき、パラメーター名には「args」を使用するのが通例だ。

辞書でキーワード引数を受け取る場合は次のようになる。

```
>>> def bar(a='foo', b='bar', **kwargs):
...   print("contents of keywords args")
...   for k, v in kwargs.items():
...     print("{0}: {1}".format(k, v))
...
>>> bar(c='hoge', d='huga')
contents of keywords args
c: hoge
d: huga
```

辞書で可変数個の引数を受け取る

辞書で受け取る場合には、そのパラメーター名の前に「**」を前置する。このとき、パラメーター名は「kwargs」にするのが通例だ。関数 bar では、辞書に対して items メソッドを呼び出して、「キー／値」の組を反復的に取り出している。また、関数 bar の呼び出しでは、パラメーター a と b がデフォルト値を持つため、省略している。よって、指定した 2 つの引数はいずれもパラメーター kwargs に渡されている。

タプルや辞書の内容を展開して、関数に渡したいというときには上で見た「*」や「**」を関数呼び出し側で利用する。以下に例を示す。

```
>>> t = (1, 2)
>>> calc(*t, '/')
0.5
>>> d = { 'a': 'foo', 'b': 'bar' }
>>> bar(**d, c='hoge', d='huga')
contents of keywords args
c: hoge
d: huga
```

タプルや辞書の値を展開して関数に渡す

次に lambda キーワードを使用した無名関数の作成と利用について見てみよう。

lambda キーワードによるラムダ式（無名関数）の定義

Python では無名関数も定義できる。これには lambda キーワードを使用して「ラムダ式」として記述する。ラムダ式の構文を以下に示す。

```
lambda パラメーターリスト: 式
```

ラムダ式の構文

これは def キーワードによる以下の関数定義とおおよそ同値である（ただし、ラムダ式で作成される関数には名前はなく、関数名は得られないので「……」としてある）。

```
def ……(パラメーターリスト):
  return 式
```

おおよそ同じ意味を持つ関数定義

ラムダ式では関数のボディーに相当する部分には単一の「式」しか記述できない。このことから分かるように、ラムダ式は極めて小さな処理を関数の形でインラインに記述する場合に使用するのが主な用途となる。ただし、以下ではまずラムダ式で作成した関数オブジェクトを変数に代入して使用してみよう。

```
>>> (lambda x, y: x + y)(1,2)
3
>>> add = lambda x, y: x + y
>>> add(2, 4)
6
```

ラムダ式による無名関数の作成

最初の例はラムダ式が返す無名関数をそのまま呼び出している。次の例では、変数 add にラムダ式が返す無名関数を代入し、その変数を利用して関数呼び出しを行っている。

次のようなこともできる。

```
>>> makeadder = lambda x: lambda y: x + y
>>> adder2 = makeadder(2)
>>> adder2(100)
102
```

ラムダ式のボディーにラムダ式を記述

一見すると分かりにくいが、最初の行は「lambda x: (lambda y: x + y)」とかっこを入れてみると少しは理解しやすくなるかもしれない。これはラムダ式を返すラムダ式ということだ。そして、返送されるラムダ式では最初のラムダ式に渡された引数にアクセスできる（よって、adder2 はパラメーター y に渡された値に、元のラムダ式のパラメーター x に渡された値= 2 を加算した値を返す関数となる）。

実際の使いどころとしてよく挙げられるのは、Python に組み込みの関数 map と組み合わせて利用することだ。関数 map はパラメーターに関数と反復可能なオブジェクトを取り、後者に対して連続的に関数を適用する反復可能オブジェクト（イテラブル、iterable）を返す。以下に例を示す。

```
>>> l = list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(map(lambda x: x * 2, l))  # 使い捨ての処理をインラインで記述
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

ラムダ式と関数 map の組み合わせ

def キーワードで定義する関数を利用しても以下のように記述してももちろん構わない。

```
>>> def dbl(x):
...     return x * 2
...
>>> list(map(dbl, l))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

関数 map に名前付きの関数を渡す

だが、上のようにちょっとした処理を行うだけの関数をわざわざ定義するのではなく、インラインでサクサクと書けるのがラムダ式（無名関数）のよいところだ（もちろん、関数 map に渡す関数でより入り組んだ処理を行うのであれば、関数を別途定義するのがよい）。

これまでの例でも出てきたが、Python では関数も他のオブジェクトと同様に、関数に引数として渡したり、関数の戻り値として返送したりできる。このような関数を引数／戻り値として扱える関数のことを「高階関数」と呼ぶ。

例えば、先ほど見たラムダ式を返すラムダ式は高階関数の一例だ。また、map 関数はパラメーターに関数を受け取るので、これも高階関数である。先ほどの例とあまり変わらないが、もう 1 つの例として Python に組み込みの関数 filter を今度は見てみよう。

```
>>> list(filter(lambda x: not x % 2, l))
[0, 2, 4, 6, 8]
```

関数 filter を利用したリストのフィルタリング

関数 filter は受け取った関数を反復可能オブジェクトの要素に適用し、その値が真となるものだけを要素とする反復可能オブジェクトを返す。Python では「0、空文字列、要素のないリスト／辞書／タプル／集合」なども偽として扱われるので、上では「x % 2」に「not」を付加することで偶数のみからなるリストを作成している。同様に関数を渡すことで、処理に介入できる関数としてはリストの sort メソッドや組み込み関数 sorted などがある。

次に関数とよく似たジェネレーターについて見てみよう。

ジェネレーター

「ジェネレーター」は「何らかの値を連続的に生成する」反復可能なオブジェクトであり、ジェネレーター関数を用いて定義する。ジェネレーターはリストやタプルなどと同様に、反復処理に利用できる（ジェネレーター関数の実際の戻り値は反復可能な「ジェネレーター」オブジェクトになる。反復処理では、ジェネレーター関数の呼び出し後に実際に制御を行うのはジェネレータオブジェクト）。

ジェネレーター関数は通常の関数と同様に def キーワードを使って定義するが（ジェネレーター式も使えるが、これについては次章で取り上げる）、ジェネレーター関数では return 文を使わずに yield 文で呼び出し側に値を返すのが大きな違いだ。yield 文が実行されると制御も呼び出し側に戻るが、関数とは異なりジェネレーターは実行時の状態を記憶しており、次回にジェネレーターへ制御が移ったときには、続きから実行が継続される。

簡単な例を以下に示す。

```
>>> def gen(n):
...   for num in range(n):
...     print("generated")
...     yield num
...
>>> for cnt in gen(10):
...   print(cnt)
...
generated
0
generated
1
generated
2
…… 省略 ……
```

単純なジェネレーター

ジェネレーター関数 gen は「引数に指定された値未満の整数列を返す」反復可能なオブジェクトを返し、その後の反復処理はこのオブジェクトを使って行われる。実行結果を見ると、呼び出し側の for ループでの数値の出力と、ジェネレーター側でのメッセージの出力が交互に行われていることから、呼び出し側の for ループとジェネレーターの間で制御が入れ替わっていることが分かる。

上の例では、有限の整数列を生成したが、無限に値を生成するジェネレーターも記述できる。例として、以下にフィボナッチ数を無限に生成し続けるジェネレーターを示す。

```
>>> def fibgen():
...   n0, n1 = 0, 1
...   yield n0  # fib(0)を返送
...   yield n1  # fib(1)を返送
...   while True:
...     n0, n1 = n1, n0 + n1  # フィボナッチ数を計算し（n0 + n1）、以前の値を1つ前の値に
...     v = yield n1
...     if v == "terminate":
...       break
```

無限にフィボナッチ数を生成し続けるジェネレーター

ここでは yield 文で変数 v に値を代入している妙な 1 行がある。実はジェネレーターを呼び出す側からは、ジェネレーターに対して send メソッドでメッセージを送信できる。送信された値は yield 文の評価結果の形で、次回に制御がジェネレーターに戻ってきたときに渡される。ここでは変数 v にその値を代入し、その値を調べてループを乱暴なやり方で終了している。実際には「StopIteration」例外が発生するので、例外処理を行うのが正しい。

このジェネレーターの利用例を以下に示す。

```
>>> fb = fibgen()
>>> for num in fb:
...   if num > 50:
...     fb.send("terminate")
...   print(num)
...
0
1
1
2
…… 省略 ……
21
34
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
StopIteration
```

フィボナッチ数ジェネレーターの呼び出し例

先ほどと異なるのは、for 文の前にジェネレーター関数 fibgen を呼び出している点だ。これにより、ジェネレーターへの参照を手に入れておき、ここではフィボナッチ数が 50 より大きくなったところで、そのジェネレーターに send メソッドでループを終了するように通知している。上の例ではメッセージの送信後、もう一度ループが実行された時点で例外が発生している。

あるいは以下のような方法もある。

```
>>> fb = fibgen()
>>> for cnt in range(10):
...   print(next(fb))
...
0
1
1
2
…… 省略 ……
```

カウンター変数を使う

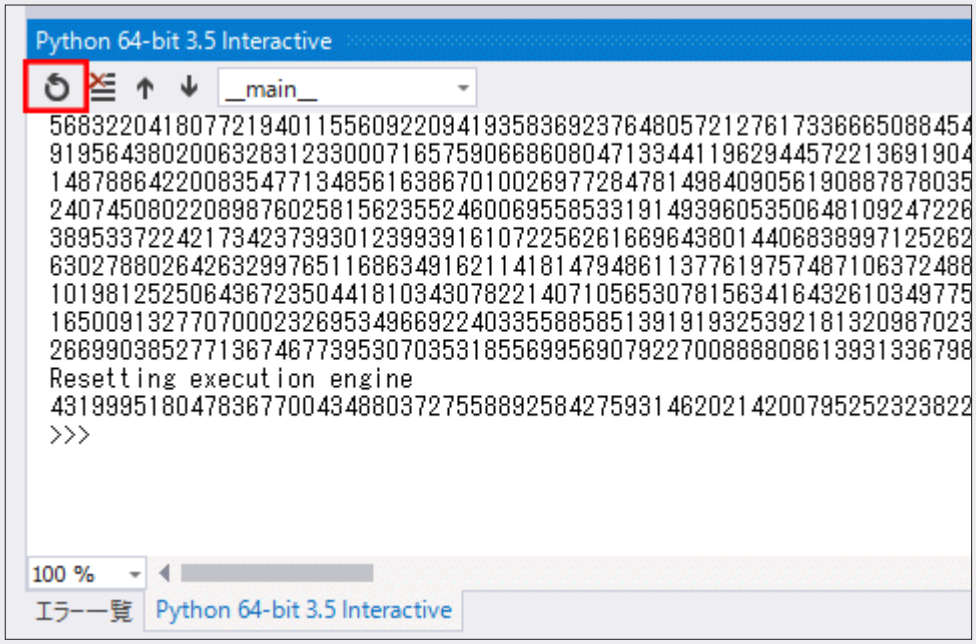
これを見ると分かるが、関数 next にジェネレーターを渡すことで順々にジェネレーターが生成する値を取得することもできる。

なお、以下のようなコードを書くと、無限ループにハマるので注意しよう。

```
>>> for num in fibgen():
...   print(num)
...
0
1
1
2
3
…… 省略 ……
```

無限ループ

このような場合には［Interactive］ウィンドウの左上にある［リセット］ボタンをクリックしよう。



［リセット］ボタン（赤枠内）で無限ループを止める

◇ ◇ ◇ ◇ ◇ ◇

本章では Python の関数とジェネレーターについて見た。次章ではジェネレーターを生成するもう 1 つの方法であるジェネレーター式とそれに関連してリストなどの内包表記などについて見ていく。

特集：Visual Studio で始める Python プログラミング

5. ジェネレーター式と内包表記を使ってみよう

本章はジェネレーター式とリストなどの内包表記を取り上げる。これらはラムダ式と並んで Python 的なコードを書くのによく使われる要素だ。

本章では、ジェネレーターを生成するもう 1 つの方法であるジェネレーター式と、それに似た構文でリストや辞書を操作できる内包表記を取り上げる。これまで同様に、VS の［Interactive］ウィンドウを利用して、動作を確認している。

ジェネレーター式

これまでのおさらいになるが、まずはジェネレーターオブジェクトを返すジェネレーター関数の定義を見ておく。

```
>>> def fibgen():
...   n0, n1 = 0, 1
...   yield n0
...   yield n1
...   while True:
...     n0, n1 = n1, n0 + n1
...     v = yield n1
...     if v == "terminate":
...       break
...
>>> fg = fibgen()
>>> for num in range(10):
...   print(next(fg))
…… 省略 ……
```

ジェネレーターの定義例

このような複雑な計算を行うジェネレーターは def キーワードを使用しないと定義できないが、シンプルなジェネレーターは「ジェネレーター式」を使っても作成できる。ジェネレーター式のシンプルな構文を以下に示す。

```
( 式 for ターゲットリスト in 反復可能オブジェクト )
```

シンプルなジェネレーター式の構文

「式」にはジェネレーターオブジェクトが返す値を、「ターゲットリスト」には「反復可能オブジェクト」（イテラブル）が返送する値を受け取るターゲット（「リスト」という表現から分かるように複数のターゲットを記述可能。受け取った値を「式」中で使用するための変数リストと考えればよい）を記述し、それらをかっこで囲む。

文章だけでは分かりにくいので、実際にシンプルな例を見てみよう。

```
>>> generator = (x * 2 for x in range(5))  # ジェネレーター式によるジェネレーターオブジェクトの作成
>>> for num in generator:
...   print(num)
...
0
2
4
6
8
```

ジェネレーター式の使用例

強調表示部分がジェネレーター式でジェネレーターオブジェクトを作成しているところだ。このジェネレーターオブジェクトは「range(5) が返す値を x に受け取り、それを 2 倍したものを返送」する。結果もその通りになっていることが分かる。この構文は、以下で取り上げるリストの内包表記などと同様であることから、「ジェネレーター内包表記」と呼ぶこともある。

なお、このジェネレーター式と同様な処理を行うジェネレーターを返すジェネレーター関数は次のようになる。

```
# 比較用 : generator = (x * 2 for x in range(5))
>>> def dblgen():
...   for x in range(5):
...     yield x * 2
```

上と同じことを行うジェネレーターオブジェクトを返すジェネレーター関数

lambda キーワードを利用したラムダ式と同様に、この構文はあまり他の言語では見かけない Python 独自のものであり、慣れないうちは分かりにくいが、使い込んでいけば便利に使えるだろう。

次に、同様な構文構造を持つリスト、集合、辞書の内包表記について見ていこう。

リスト、集合、辞書の内包表記

内包表記とは既に存在している何らかのデータ（シーケンスや反復可能オブジェクト）を基にして、新たなデータを作成するための簡便な記述法だと考えられる。ジェネレーター式と大きく異なるのは、「式 for ターゲットリスト in 反復可能オブジェクト」を囲むものだ。リストでは「[]」で、集合と辞書では「{}」で囲む。辞書ではさらにキー／値を表現するために「式」の部分にコロン（:）で区切った式を 2 つ記述することになる。後述するがタプルには内包表記はない。

```
# リストの内包表記
[ 式 for ターゲットリスト in 反復可能オブジェクト ]

# 集合の内包表記
{ 式 for ターゲットリスト in 反復可能オブジェクト }

# 辞書の内包表記
{ 式1 : 式2 for ターゲットリスト in 反復可能オブジェクト }
```

リスト／集合／辞書の内包表記構文

内包表記は、式（あるいは「式 1 : 式 2」）を評価した結果を要素とする新たなリスト／集合／辞書を作成する。これに対して、ジェネレーター式が返すのはジェネレーターオブジェクトである点は大きな違いだ（内包表記では実際にリストなどが作成されメモリがその要素に必要なだけ占有されるが、ジェネレーター式では全要素に必要なメモリが占有されることはない）。

以下では基本要素を見ていく。まずはリストの内包表記から。

```
>>> l1 = list(range(10))
>>> l1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l2 = [x * 2 for x in l1]  # リストの内包表記
>>> l2
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>> l3 = [x * 2 for x in range(10)]  # もちろん、こう書いてもよい
```

内包表記によるリストの作成

「{}」で囲むことを除けば、集合も同様だ。

```
>>> s1 = set(range(0, 10, 2))
>>> s1
{0, 8, 2, 4, 6}
>>> s2 = {x * 2 for x in s1}  # 辞書の内包表記
>>> s2
{0, 16, 4, 12, 8}
>>> s3 = {x * 2 for x in range(0, 10, 2)}
>>> s4 = {x * 2 for x in l1}  # リストを基に集合を作成
```

内包表記による集合の作成

辞書については先にも述べたように、キーと値が必要になるので、少し書き方が変わる。ここでは「式」の部分が「n : chr(n + 65)」となっている。chr 関数は引数に整数を取り、それを Unicode のコードポイントとして、対応する 1 文字の文字列を返すので、変数 d の内容を見ると分かるように A ～ E の各文字が得られている。

```
>>> d = { n : chr(n + 65) for n in range(5) }  # 辞書の内包表記
>>> d
{0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E'}
```

内包表記による辞書の作成

なお、タプルには内包表記がない。上で見た通りジェネレーター式で「()」を使用しているので、「(x for x in ...)」と書いてもこれはタプルの内包表記ではなく、ジェネレーター式となる。タプルが欲しい場合には、単純に組み込み関数 tuple の引数に内包表記形式の式を書けばよい。このとき、内包表記を囲むかっこは省略できる。なお、リストや集合、辞書も同様に組み込み関数 list ／ set ／ dict に内包表記を表記できるが、関数 dict に関してはかっこを省略できないので注意しよう。以下に例を示す。

```
>>> tuple(x * x for x in range(5))     # 内包表記を関数tupleの引数として記述
(0, 1, 4, 9, 16)
>>> list(x * x for x in range(5))      # リストの場合
[0, 1, 4, 9, 16]
>>> set(x * x for x in range(5))       # 集合の場合
{0, 1, 4, 9, 16}
>>> dict(x : x * x for x in range(5))  # 辞書では{}を省略できない
  File "<stdin>", line 1
    dict(x : x * x for x in range(5))
           ^
SyntaxError: invalid syntax
>>> dict({x : x * x for x in range(5)})
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

組み込み関数に内包表記を渡すことでタプル／リスト／集合／辞書を作成

少し高度な内包表記の例

前ページで見た内包表記は非常にシンプルなものだったが、二重の for ループに相当する処理を内包表記で表現したり、「式」を適用する前に要素のフィルタリングを行ったりすることもできる。以下では、少し高度な内包表記の使い方を見てみよう。

for 節のネスト

まずは以下の for ループを見てみよう。

```
>>> l = []
>>> for x in range(3):
...   for y in range(4):
...     l.append((x, y))
...
>>> l
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), …… 省略 ……, (2, 2), (2, 3)]
```

for 文の二重ループ

→目次に戻る

このコードでは for 文を二重にループさせて、(0, 0) ～ (2, 3) のタプルをリストに追加している。これと同等な処理を内包表記で記述すると次のようになる。

```
>>> l = [(x, y) for x in range(3) for y in range(4)]
>>> l
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), …… 省略 ……, (2, 2), (2, 3)]
```

内包表記による二重ループの表現

このように「for 節」（for ターゲットリスト in 反復可能オブジェクト）をネストさせることで、二重ループをするよりも簡潔な形でコードを記述できる。for 文を使った場合とループと同じ順序で for 節をネストさせればよいので、for ループのネストから for 節のネストへの頭の切り替えはそれほど難しくはないだろう。このことが手や体や頭に染みついてきたら、次の内包表記の意味も分かるようになる。

```
>>> [num for item in [[1], [2, 3], [4, 5, 6]] for num in item]
[1, 2, 3, 4, 5, 6]
```

リストを要素とするリストのフラット化

ネストした for 節はネストした for 文と同じ順序で書かれるので、これはおおよそ次のような for ループと同等だと考えればよい。

```
for item in [[1], [2, 3], [4, 5, 6]]:
  for num in item:
    # numをリストの要素に追加
```

内包表記の for 文への展開

一見すると難解な表現でも、最初のうちは自分が理解できている構文要素へ展開してみると「ああ、そういうことか」となるので、つまずいたときには基本に戻ってみるようにしよう。

if 節による要素のフィルタリング

for 節に続けて「if 節」を記述すると、元の反復可能オブジェクトから要素をフィルタリングできる。以下に例を示す。

```
>>> l = ["Windows Server Insider", "Insider.NET", "Server & Storage", "HTML5 + UX"]
>>> l2 = [item for item in l if "Insider" in item]  # "Insider"が含まれる要素からなるリストを作成
>>> l2
['Windows Server Insider', 'Insider.NET']
>>> s = {x * x for x in range(10) if x % 2 == 0}  # 偶数のみを取り出し、その二乗からなる集合を作成
>>> s
{0, 16, 64, 4, 36}
>>> s = {x * x for x in range(10) if x % 2}  # 奇数のみを取り出し、その二乗からなる集合を作成
>>> s
{81, 1, 49, 9, 25}
```

if 節によるフィルタリング

for 節がネストしている場合、if 節はそれぞれに付加できる。以下は range(5）の範囲で x には奇数のみ、y には偶数のみを取り出して、それらを組み合わせたタプルを要素とするリストを作成している。

```
>>> [(x, y) for x in range(5) if x % 2 for y in range(5) if not y % 2]
[(1, 0), (1, 2), (1, 4), (3, 0), (3, 2), (3, 4)]
```

2 つの for 節のそれぞれに if 節を追加

なお、Python では三項演算子（条件演算子）の代わりに条件式が使える。以下に構文を示す。

```
式1 if 条件 else 式2
```

Python の条件式

この式ではまず「条件」が評価される。その値が真であれば「式 1」の値が、偽であれば「式 2」の値がその式の評価結果となる。そのため、else 以降を省略することはできない。

使用例を以下に示す。以下は乱数が偶数か奇数かでその値が変化する条件式だ。

```
>>> import random
>>> 100 if random.randint(0,10) % 2 else 200
100
>>> 100 if random.randint(0,10) % 2 else 200
200
```

条件式の使用例

この条件式を「式」部分に記述することも可能だ。for 節に続ける if 節とは異なるので注意が必要だ。強調表示した部分が内包表記の「式」に当たる。

```
>>> l = ["Windows Server Insider", "Insider.NET", "Server & Storage", "HTML5 + UX"]
>>> [item if "Insider" in item else "outsider" for item in l]
['Windows Server Insider', 'Insider.NET', 'outsider', 'outsider']
```

条件式の使用例

zip 関数との組み合わせ

最後に zip 関数を使った辞書の作成を見てみる。zip 関数は複数の反復可能オブジェクトを受け取り、それらの要素を一組にしたタプル（のイテレータ）を返送する。これを利用すると、例えば、以下のような辞書を作成できる。

```
>>> l = ["Windows Server Insider", "Insider.NET", "Server & Storage", "HTML5 + UX"]
>>> {"forum {0}".format(n) : forum for n, forum in zip(range(1,5), l)}
{'forum 2': 'Insider.NET', 'forum 4': 'HTML5 + UX', 'forum 3': 'Server & Storage', 'forum
1': 'Windows Server Insider'}
```

zip 関数と組み合わせて、辞書を作成

◇ ◇ ◇ ◇ ◇ ◇

本章ではジェネレーター式と、それとよく似た構文を持つリスト／集合／辞書の内包表記について見てきた。次章からはモジュールとクラスなどを取り上げる。

特集：Visual Studio で始める Python プログラミング

6. Python のクラス、最速理解

本章ではシンプルなクラス定義、各種のメンバを持つクラスの定義、継承と多重継承の方法など、Python 3 のクラスについて駆け足で見ていこう。

これからの幾つかの章ではクラスとモジュールについて見ていこう。動作は VS の［Interactive］ウィンドウを使用して確認している。

シンプルなクラスの定義

Python で最もシンプルなクラス定義は次のようになる。

```
class Foo:
  pass
```

シンプルなクラス

「pass」文は何もしないことを表す文だ。Python では「class ～:」の後にブロックがないとエラーとなるので、何もしないことを示すために pass 文を記述する（C# などの言語における「{}」と似たようなものといえる）。

上記のように「基底クラス」を指定していない場合、そのクラスは object 型を基底クラスとする。

クラスのインスタンスを作成するには、C# などの言語とは異なり「new」は不要で、以下のようにクラス名に続けてかっこを記述して関数的に使用する。

```
f = Foo()  # インスタンス生成に「new」キーワードは不要
print(type(f))    # 出力結果： <class '__main__.Foo'>
```

インスタンスの生成

組み込み関数 type の出力結果を見ると分かるように、変数 f の型は「__main__.Foo」となっている。Python の「ライブラリリファレンス」によると、「__main__」は「トップレベルのコードが実行されるスコープの名前」である。簡単にいうと、［Interactive］ウィンドウのような対話環境でプログラマーが Python コードを直接入力したり、コマンドラインから「python ～.py」を入力して Python スクリプトを実行したりすると、それらのコードは「__main__」という組み込みのスコープで実行される。よって、「<class '__main__.Foo'>」はそのスコープ内で定義されたクラス Foo ということになる。

なお、スコープやモジュールについては次章で詳しく説明する。

メンバを持つクラスの定義

もちろん、クラスにはメンバがなければ役には立たない。以下にメンバを持つクラス定義がどうなるか、その構文を示す。

```
class クラス名:
  クラス変数名 = ...  # クラス変数

  # インスタンスの初期化
  def __init__(self[, パラメーターリスト]):  # インスタンスの初期化を行う
    self.インスタンス変数名 = ...  # インスタンス変数の初期化を行う

  # インスタンスメソッド
  def instance_method(self[, パラメーターリスト]):
    インスタンスメソッドの実装

  @classmethod  # クラスメソッドは「@classmethod」で修飾する
  def class_method(cls[, パラメーターリスト]):
    クラスメソッドの実装

  @staticmethod  # スタティックメソッドは「@staticmethod」で修飾する
  def static_method([パラメーターリスト]):
    スタティックメソッドの実装
```

Python 3 でのクラス定義

見慣れないのは「__init__」という表記だ。__init__ メソッドは、インスタンスの初期化に使われるメソッドであり（上の例であれば、「f = Foo()」としたときに __init__ メソッドがあれば、インスタンスが生成された後でそれが呼び出される）、他のプログラミングにおけるコンストラクタに近い処理を行うものだ。

__init__メソッドとインスタンスメソッド（上では instance_method）の第 1 パラメーターには慣例として「self」を指定する。__init__ メソッドでは「self. インスタンス変数名 = ...」のように self を使用して、そのインスタンスが持つインスタンス変数の初期化を行う。また、「インスタンス . メソッド (引数)」のようにしてインスタンスメソッドを呼び出すと、メソッド呼び出しに使われたインスタンスが暗黙の引数として self に渡される。その後ろの「パラメーターリスト」が実際のメソッド呼び出しにおける引数を受け取る。

__init__ メソッドとインスタンスメソッドを持つクラス

まずは、__init__ メソッドとインスタンスメソッドを持つクラスを定義してみよう *1。

*1　なお、以降のコードではメソッド定義間に空行を挟んでいる。そのため、これらのコードを［Interactive］ウィンドウに見た通りに入力していくと、空行を入力した時点でクラス定義が終了してしまう。手入力しようという場合には空行を入力しないように注意してほしい。

```
class Foo:
  def __init__(self, name):
    self.name = name

  def hello(self):
    print("hello", self.name)

f = Foo("insider.net")
f.hello()
```

インスタンス変数とインスタンスメソッドを持つクラス

これを実行すると次のようになる。

```
>>> class Foo:
...   def __init__(self, name):
...     self.name = name
...
...   def hello(self):
...     print("hello", self.name)
...
>>> f = Foo("insider.net")
>>> f.hello()
hello insider.net
```

上記プログラムの実行結果

インスタンスメソッドの内部でインスタンス変数にアクセスするには「self.」を前置する必要があるのは、C# に慣れている方には注意が必要かもしれない。C# ではインスタンス変数へのアクセスには「this」を省略できることが多いが、Python ではそうはいかない。あるインスタンスメソッドから、別のインスタンスメソッドを呼び出す際も同様だ。上記のクラス Foo にメソッドを増やしてみよう。

```
class Foo:
  def __init__(self, name):
    self.name = name

  def hello(self):
    print("hello", self.name)

  def bar(self):
    print("instance method #2")
    self.hello()

f = Foo("insider.net")
f.hello()
f.bar()
```

インスタンスメソッド bar を追加

インスタンスメソッド bar からは同じクラスのインスタンスメソッド hello を呼び出しているが、ここでも「self.hello()」のように「self.」が必要となる。「self.」を抜いて上記コードを実行した結果を以下に示す（「self.」付きの例は省略）。

```
>>> class Foo:
...   def __init__(self, name):
...     self.name = name
...
...   def hello(self):
...     print("hello", self.name)
...
...   def bar(self):
...     print("instance method #2")
...     hello()  # 「self.」を抜いてみた
...
>>> f = Foo("insider.net")
>>> f.hello()
hello insider.net
>>> f.bar()
instance method #2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in bar
NameError: name 'hello' is not defined
```

「hello」という名前が見つからない

インスタンスメソッドのパラメーターリストの先頭には「self」が必要であることと、インスタンス変数やインスタンスメソッドのアクセスには「self」を介する必要があることは覚えておこう（この名前は慣例的なものだが、他の Python プログラムを読めば、まずそうなっているし、自分のコードを他の Python プログラマーが読むときのことを考えると、「self」とするのが強く推奨される）。

クラス変数、クラスメソッド、スタティックメソッド

上でも示したようにクラス変数は「class クラス名 :」の直下のブロックに記述する。名前から分かるように、これはクラスレベルの変数であり、複数のインスタンスで共有される。クラスメソッドも同様にクラスレベルのメソッドとなる。クラス変数とクラスメソッドを持つクラスの例を以下に示す。これはインスタンスが生成されるごとに、クラス変数 _count をカウントアップしていき、クラスメソッド count を呼び出すとそのカウントを表示するクラスだ。

```
class Bar:
  _count = 0

  def __init__(self):
    Bar._count += 1

  @classmethod
  def count(cls):
    print(cls._count)

Bar.count()
b = Bar()
b.count()
Bar.count()
b = Bar()
Bar.count()
```

クラス変数とクラスメソッドを持つクラス

クラス変数「_count」の「_」はクラスメソッド count の名前との重複を避けるという意味もあるが、Python では「_」を 1 つだけ前置した識別子は「内部で使用する」ことを意味するというスタイルガイドに合わせたものだ。Python には C# などの言語ではよく使われている「プライベート」という概念が存在せず、全てのメンバは基本的に外部に対して公開されているが、コードを記述する側が「これは内部で使用するもの」と他者に伝えるためにこのような使い方が推奨されている。

__init__ メソッドでは「クラス名 . クラス変数名」という形式でクラス変数 _count にアクセスし、カウントアップをしている。クラスメソッド count はクラスオブジェクト（クラスを表すオブジェクト）を第 1 パラメーターに受け取るので、「cls._count」のような形でクラス変数にアクセスをしている。

クラス定義の下では、インスタンスを生成して、クラスメソッド count を呼び出している。見れば分かるように、クラスメソッドは「クラス名 . クラスメソッド名 (...)」と「インスタンス名 . クラスメソッド名 (...)」のどちらの方法でも呼び出せる（ただし、インスタンス経由でクラスメソッドを呼び出しても、呼び出しで重要なのはそのインスタンスの型だけであり、インスタンスが保持する情報はクラスメソッドでは利用されない）。

以下に実行結果を示す。

```
>>> class Bar:
...   _count = 0
...
...   def __init__(self):
...     Bar._count += 1
...
...   @classmethod
...   def count(cls):
...     print(cls._count)
...
>>> Bar.count()
0
>>> b = Bar()
>>> b.count()
1
>>> Bar.count()
1
>>> b = Bar()
>>> Bar.count()
2
```

実行結果

今見たように、クラスメソッドはクラスオブジェクトを受け取り、その情報を利用できたが、スタティックメソッドは第 1 パラメーターにクラスオブジェクトを受け取らないのが大きな違いだ。

```
class Bar:
  _count = 0

  def __init__(self):
    Bar._count += 1

  @classmethod
  def count(cls):
    print(cls._count)

  @staticmethod
  def static_method():
    print("static method")

Bar.static_method()
b = Bar()
b.static_method()
```

スタティックメソッドの例

第 1 パラメーターにクラスオブジェクトを受け取らないことから、スタティックメソッドはクラスをモジュール／名前空間とした何らかのユーティリティー関数を実装したりするのに使えるだろう（実行例は割愛）。

プロパティ

setter と getter を伴うプロパティも定義可能だ。これには組み込み関数 property を使用する。以下に例を示す。

```
class Baz:
  def __init__(self, name):
    self._name = name

  def getname(self):  # getter
    return self._name

  def setname(self, value):  # setter
    self._name = value

  name = property(getname, setname)  # プロパティの設定

b = Baz("hogehoge")
print(b.name)
b.name = "insider.net"
print(b.name)
```

プロパティの定義

このように getter ／ setter となるメソッドを定義して、それを組み込み関数 property に渡してやればよい。setter を指定しなければ、それは読み取り専用のプロパティとなる。実際には組み込み関数 property はパラメーターを 4 つ取る。

```
property(fget=None, fset=None, fdel=None, doc=None)
```

組み込み関数 property の構文

パラメーター fdel にはプロパティを削除するメソッドを、doc にはそのプロパティについてのドキュメントを指定する。

あるいは @property デコレータとして各メソッドを修飾してもよい。先ほどのコードを @property デコレータを用いて書き直したものを以下に示す。

```
class Baz:
  def __init__(self, name):
    self._name = name

  @property
  def name(self):
    return self._name

  @name.setter
  def name(self, value):
    self._name = value
```

デコレータ形式でのプロパティ定義

この場合は getter となるメソッドに「@property」で修飾をしてメソッド名をプロパティ名にする（この場合は「getname」ではなく「name」）。また、setter となるメソッドには「@ プロパティ名 .setter」と修飾する。メソッド名はプロパティ名と同じものにする（この場合は「setname」ではなく「name」）。

先ほども述べたが、Python には「プライベート」という概念がなく、全てのメンバが基本的には外部に公開される。さらに言えば、以下のようにクラス定義の段階では存在しないメンバ（インスタンス変数／データ属性）をインスタンスに付加することも可能だ。このコードでは name プロパティが読み取り専用になっていることに注意（@name.setter 修飾されたメソッドがない）。

```
class Baz:
  def __init__(self, name, addr):
    self._name = name
    self.addr = addr

  @property
  def name(self):      # name.setter修飾されたメソッドがないので、
    return self._name  # nameは読み取り専用プロパティ

b = Baz("insider.net", "tokyo")
b.addr = "yokohama"
b.prop = "build insider"  # 変数bが参照するオブジェクトにインスタンス変数を追加
print(b.addr)
b.method = lambda x: print("b.method:", x)  # メソッドも追加できる
b.method("hoge")
b.name = "windows server insider"  # エラー：nameプロパティは読み取り専用
```

インスタンスに対しては自由にデータを追加したり、その値を変更したりできるが、プロパティなら自由な変更を許さないようにもできる

この例では、Baz クラスのインスタンスに対して、インスタンス変数を追加したり（b.prop）、インスタンスメソッドを追加したり（b.method）、クラス定義の時点で存在しているインスタンス変数の値を自由に変更したりしている。その一方で、b.name は読み取り専用のプロパティとなっているので、これを変更はできない。プロパティを使うことで、クラスのメンバを読み取り／更新するための適切なインタフェースを提供できるようになる。

Python の命名規約

Python のコーディングスタイルガイド（日本語訳はこちら）では、クラス名は「CapWords」形式とある。これは「単語の先頭を大文字にして、複数の単語はそのまま連結する」ことを意味する。

関数名は「小文字だけを使い、可読性を高める必要があれば、複数の単語はアンダースコアでつなぐ」ことが基本的には推奨されている。インスタンスメソッドの最初のパラメーターには既に見たように「self」を、同じくクラスメソッドの最初のパラメーターには「cls」を使うことも推奨されている。

この他にも上で述べた「_」の使い方をはじめとする Python で推奨されている命名規約、コーディングスタイルが掲載されているので、ざっと目を通しておいた方がよいだろう。

クラスの継承

本章の冒頭でも述べた通り、基底クラスを指定しない場合、クラスは object クラスの派生クラスとなる。基底クラスを指定してクラスを定義するには、次のように「class 派生クラス名 (基底クラス名のリスト):」のように基底クラスを指定する。

```
class 派生クラス名(基底クラス名のリスト):
  クラス定義
```

クラスの継承

「既定クラス名のリスト」となっていることから分かる通り、Python では多重継承がサポートされている。

まずはシンプルな単一継承の例を見てみよう。

```
class Base:
  def __init__(self, base_mem):
    self.base_mem = base_mem

class Derived(Base):
  def __init__(self, base_mem, derived_mem):
    super().__init__(base_mem)
    self.derived_mem = derived_mem

d = Derived("base", "derived")
print(d.base_mem)
print(d.derived_mem)
```

クラスの継承

上で述べたように Derived クラスでは「class Derived(Base):」として、基底クラスが Base クラスであるこ

とを示している。重要なのは、Derived クラスの __init__ メソッド内だ。基底クラスの __init__ メソッドを「super().__init__(base_mem)」という形で呼び出している（組み込み関数 super で基底クラスを参照し、その __init__ メソッドを呼び出しているということだ）。Python では組み込み関数 super は他にも多くの場面で使用される。例えば、オーバーライドされたメソッドから基底クラスの同名のメソッドを呼び出す場合などがそうだ。というわけで、次にメソッドのオーバーライドを見てみよう。

メソッドのオーバーライド

先ほどの Base クラスと Derived クラスにインスタンスメソッド method を追加したコードを以下に示す。

```
class Base:
  def __init__(self, base_mem):
    self.base_mem = base_mem

  def method(self):
    print("method on Base")

class Derived(Base):
  def __init__(self, base_mem, derived_mem):
    super().__init__(base_mem)
    self.derived_mem = derived_mem

  def method(self):
    print("method on Derived")

d = Derived("base", "derived")
d.method()
```

インスタンスメソッドのオーバーライド

この場合、Base クラスのインスタンスメソッド method は Derived クラスのインスタンスメソッド method に隠蔽される。よって結果は次のようになる。

```
>>> class Base:
...   def __init__(self, base_mem):
…… 省略 ……
...
>>> d = Derived("base", "derived")
>>> d.method()
method on Derived
```

実行結果

Derivedクラスのインスタンスメソッドmethodから基底クラスの同名メソッドを呼び出したい場合には次のようにする。

```
class Derived(Base):
  def __init__(self, base_mem, derived_mem):
    super().__init__(base_mem)
    self.derived_mem = derived_mem

  def method(self):
    super().method()
    print("method on Derived")
```

基底クラスの同名メソッドを呼び出す

先ほどの __init__ メソッドと同様に「super().method()」として基底クラスのインスタンスメソッド method を呼び出せる。これはクラスメソッドでも同様だ。

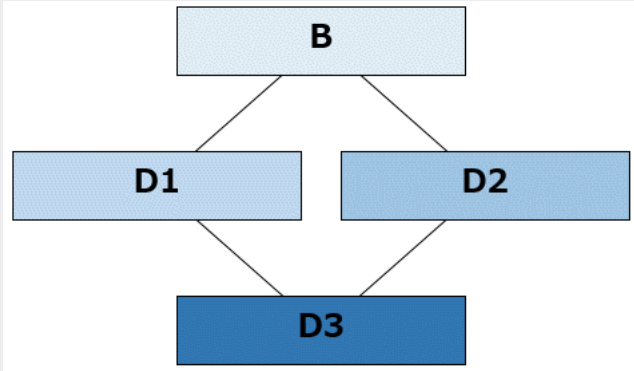
多重継承

多重継承を行う場合には、クラス定義時に基底クラスとなるクラスを列挙していく。以下に例を示す（クラス名やメソッド名は上の例よりもシンプルにした）。

```
class B:
  def m(self):
    print("m on B")

class D1(B):
  def m(self):
    print("m on D1")

class D2(B):
  def m(self):
    print("m on D2")

class D3(D1, D2):
  pass

d3 = D3()
d3.m()
```

D3 クラスは D1 クラスと D2 クラスを継承する

これは典型的なダイヤモンド継承だ。



ダイヤモンド継承

ダイヤモンド継承においては、メソッド呼び出しをどう解決するかが問題になる（Python 3 では object クラスをルートとしたクラス階層の中に全てのクラス／オブジェクトが含まれるため、上のコードのようにしなくとも、おのずとダイヤモンド継承にまつわる問題が発生する）。D3 クラスではインスタンスメソッド m を定義（オーバーライド）していないが、これを呼び出すとどのメソッドが呼び出されるだろうか。

```
>>> class B:
...   def m(self):
...     print("m on B")
...
>>> class D1(B):
...   def m(self):
...     print("m on D1")
...
>>> class D2(B):
...   def m(self):
...     print("m on D2")
...
>>> class D3(D1, D2):
...   pass
...
>>> d3 = D3()
>>> d3.m()
m on D1
```

D1 クラスのインスタンスメソッド m が呼び出された

ここでは、D1 クラスのメソッドが呼び出された。次にもう一度、実験をしてみる。D3 クラスの定義での基底クラス指定を「(D2, D1)」にした以外は同じものを実行した結果を以下に示す。

```
…… 省略 ……
>>> class D3(D2, D1):
...   pass
...
>>> d3 = D3()
>>> d3.m()
m on D2
```

今度は D2 クラスのメソッドが呼び出された

→目次に戻る

基底クラスの指定順序を変えると、呼び出されるメソッドも変化するということだ。

どのメソッドが呼び出されるのかは、クラス階層をさかのぼってメソッドや属性を探索する順序による。詳細は Python における「新スタイルクラス」（Python 3 のクラスはこのスタイルのクラス）のメソッド解決順序（MRO: Method Resolution Order）を定めた「The Python 2.3 Method Resolution Order」が詳しいのだが、こまごまとした規則を覚えていなくともクラスオブジェクトの __mro__ 属性を使えば、これを取得できる（MRO は「深さ優先、左から右、の順番で検索をするが、検索ルートの中で特定のクラスが複数回出てきた場合には後回しとなるよう」に基本的には決定される。「後回し」とは複数ある直接基底クラスが共通する基底クラスを持っていた場合、その基底クラスは直接基底クラスよりも検索順が後になるということだ。このため、一見すると幅優先に近い順序で検索が行われるように見える。ただし、実際にはより複雑な処理が行われているようだ）。

例えば、変更前のクラスの「D3.__mro__」属性の値は次のようになる。

```
>>> D3.__mro__
[<class '__main__.D3'>, <class '__main__.D1'>, <class '__main__.D2'>,
<class '__main__.B'>, <class 'object'>]
```

最初の D3 クラスでは D3 → D1 → D2 → B → object の順に検索が行われる

これに対して、基底クラスの指定を「(D2, D1)」に変更した D3 クラスでは「D3.__mro__」属性の値は次のようになる。

```
>>> class D3(D2, D1):
...   pass
...
>>> D3.mro()
[<class '__main__.D3'>, <class '__main__.D2'>, <class '__main__.D1'>,
<class '__main__.B'>, <class 'object'>]
```

変更後の D3 クラスでは D3 → D2 → D1 → B → object の順に検索が行われる

また、組み込み関数 super を使うと多重継承時のメソッド探索をカスタマイズできる（単一継承時には上でも見たようにオーバーロードされたメソッドから基底クラスのメソッドを呼び出すのに使える。ただし、単一継承時でも MRO によって検索が行われることには変わりはない。多重継承では MRO の決定が単一継承よりも複雑になるということだ）。例えば、上の D3 クラスは次のように変更できる。

```
…… 省略 ……
class D3(D1, D2):  # 基底クラスの指定はD1、D2の順序
  def m(self):
    super(D3, self).m()

d3 = D3()
d3.m()
```

D3 クラスにインスタンスメソッド m を追加

→目次に戻る

Python のライブラリレファレンスによれば、この形式の組み込み関数 super は「メソッドの呼び出しを type の親または兄弟クラスに委譲するプロキシオブジェクトを」返す（type は第 1 引数。この場合は D3）。つまり、D3 の親か兄弟クラスのインスタンスメソッド m を呼び出すということだ。第 2 引数は、インスタンスメソッド呼び出しで使われるコンテキスト（オブジェクト）だ。つまり、D3 クラスのインスタンスに対して、親クラスまたは兄弟クラスのインスタンスメソッド m を呼び出すという意味になる。なお、引数を省略した「super().m()」は「super(D3, self).m()」と同じ意味になる。

この場合には D3.__mro__ 属性の値に従ってメソッドの検索は D1 → D2 → B の順に行われる（「親または兄弟クラス」という点に注意。D3 自体は検索対象からは外れる）。よって、「d3.m()」呼び出しの結果は次のようになる。

```
>>> d3 = D3()
>>> d3.m()
m on D1
```

D1 クラスのメソッドが呼び出される

今度は D3 クラスを次のように変更してみる。

```
class D3(D1, D2):
  def m(self):
    super(D1, self).m()

d3 = D3()
d3.m()
```

D1 の「親または兄弟」となるクラスにインスタンスメソッド m の呼び出しを委譲する

実行結果は次のようになる。

```
>>> d3 = D3()
>>> d3.m()
m on D2
```

今度は D2 クラスのメソッドが呼び出される

ソースコードまで追えなかったので推測になるが、これにより、D3.__mro__ 属性の値で D1 よりも後ろにあるもの（つまり、親または兄弟となるクラス）に対してメソッド解決が行われるようになると思われる。

多重継承を行った場合のメソッド呼び出しの解決はプログラマーの頭を悩ませるものだが、Python では明確なルール（MRO）とそのカスタマイズ手段が与えられている。とはいえ、シンプルさを重要視するのであれば、本稿では触れられなかったがミックスインなどの手法を採用するのがよいかもしれない。

◇ ◇ ◇ ◇ ◇ ◇

本章では Python のクラスについてざっくりと一巡りしてきた。次章ではモジュールについて見ていく。

特集：Visual Studio で始める Python プログラミング

7.Python のモジュールの基本を押さえる

本章ではモジュールの作成、インポート、グローバル変数 __name__、コマンドラインからのモジュールの実行など、Python のモジュールの基礎概念を取り上げる。

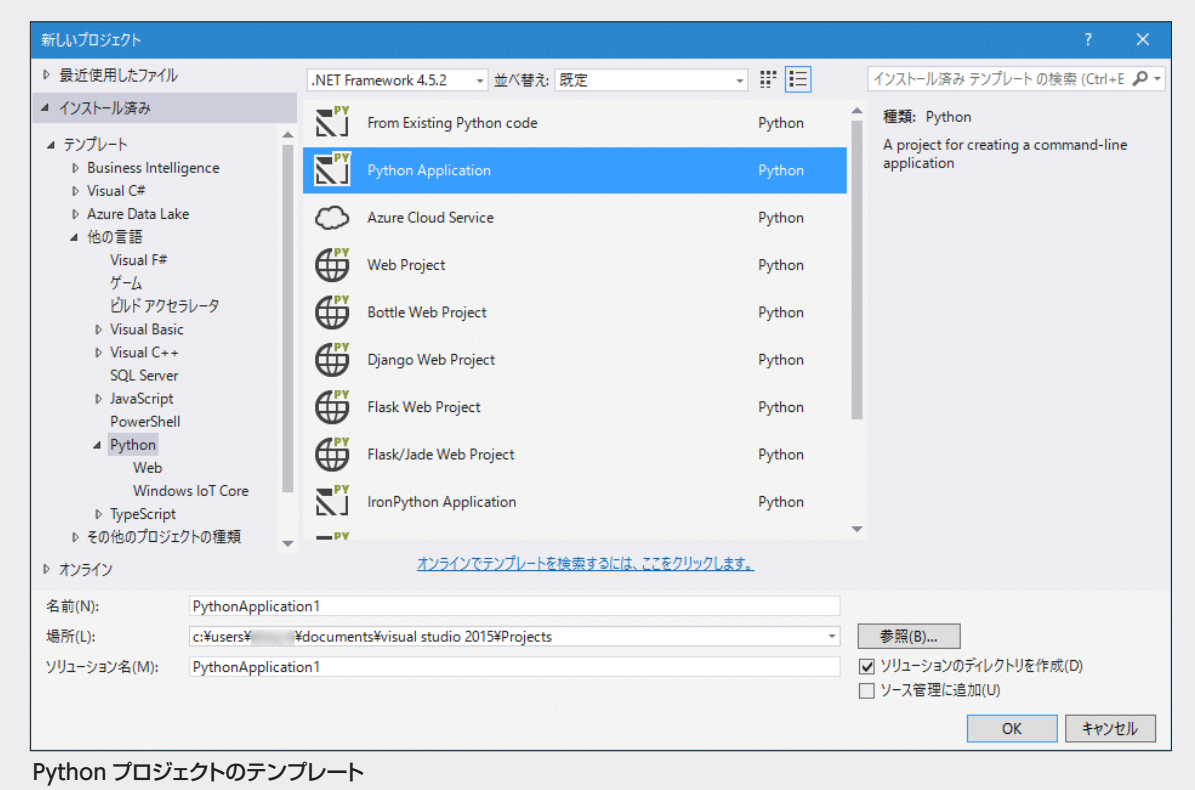
前章では Python のクラスについて解説した。例外処理など、まだ説明していない要素も多いが、基本的な構文要素や関数、クラスなどについてはおおよその紹介が終わったので、本章ではコードの再利用を促すためのキーパーツである「モジュール」について見ていこう。

モジュールとは

Python における「モジュール」とは「独立したファイルに収められた Python スクリプト」のことだ。そういうわけで、前章までは VS の［Interactive］ウィンドウでの対話的なコードの入力／実行を前提として解説をしてきたが、本章では VS で Python プロジェクトを作成してみよう。

Python プロジェクトの作成

VS の［新しいプロジェクト］ダイアログを開いて、開発言語として Python を選択すると次のような画面になる（以下は PTVS をインストールした VS 2015 のもの。VS 2017 でも同様だ）。



Python プロジェクトのテンプレート

→目次に戻る

Python 用のプロジェクトテンプレートには、既存の Python コードから VS 用のプロジェクトを作成するもの、Python のコンソールアプリ用のテンプレート、Azure や Web アプリ用のもの、IronPython 用のテンプレートなどがある。ここでは、モジュールの話をしたいだけなので、シンプルに上から 2 番目にある［Python Application］テンプレート（あるいは［Python アプリケーション］）を利用してプロジェクトを作成しよう。プロジェクト名はデフォルトのままとした（ここでは「PythonApplication1」）。

これにより、VS 用の Python プロジェクトが作成される。実際に作成されるのは、PythonApplication1.py ファイルと PythonApplication1.pyproj ファイルの 2 つだけだ。前者には Python コードを記述していく。後者には VS で Python プロジェクトを管理するために必要な情報が記述される（実際には、コマンドプロンプトを開いて「python PythonApplication1.py」コマンドを実行するだけで、スクリプトをコマンドラインから実行できる。もちろん、Python のインストール／パス設定などは必要だ）。

VS に Python サポートをインストールしていれば、デバッグ実行もボタン一発で済み、作成したモジュールを［Interactive］ウィンドウで試しに使ってみるのも簡単だ。そのため、以下では VS での作業を念頭に話を進めるが、好きなエディタとコマンドプロンプト（Python の対話環境および python コマンドを実行する）を使っても本稿の内容は試せるはずだ。

では、実際に PythonApplication1.py ファイルに簡単なコードを書いてみよう。

モジュールの作成

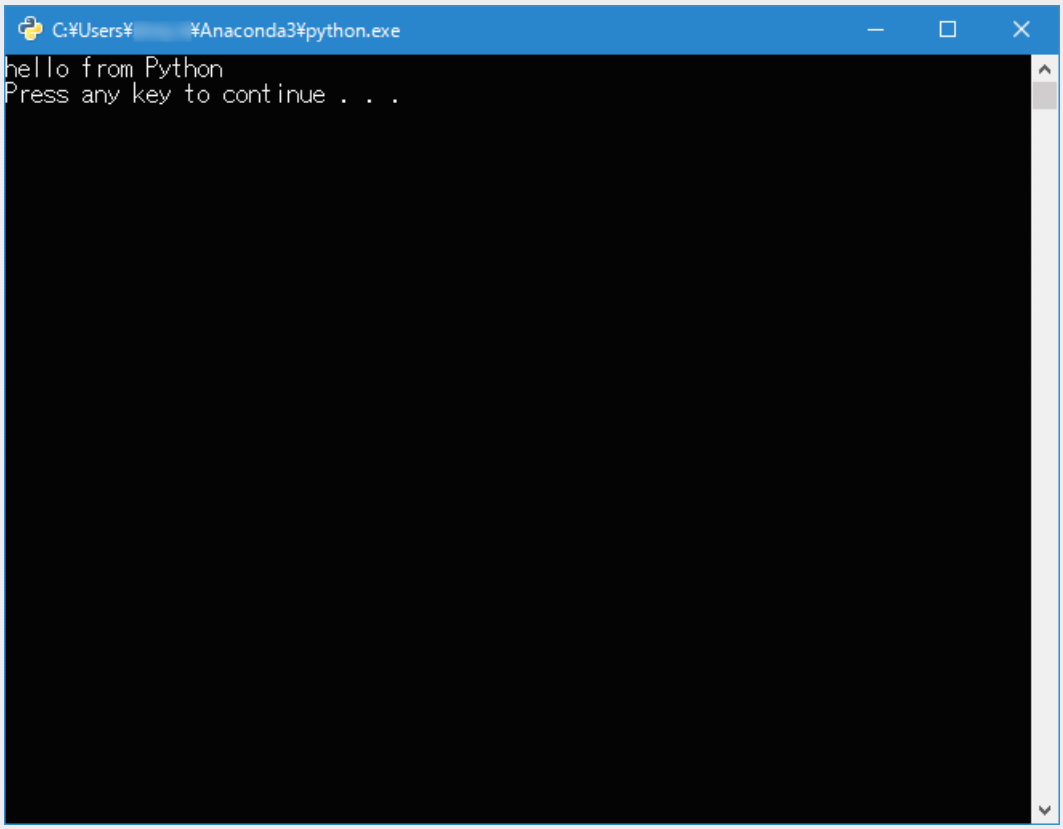
先ほども述べたように、Python におけるモジュールとは独立したファイルに記述された Python コード群のことだ。そこで、今作成したプロジェクトに含まれる PythonApplication1.py ファイルにコードを記述すれば、Python モジュールが作られるということだ。ここでは以下のようなコードを記述してみる。

```
print("hello from Python");

def hello(name):
  print("hello", name)
```

初めてのモジュール

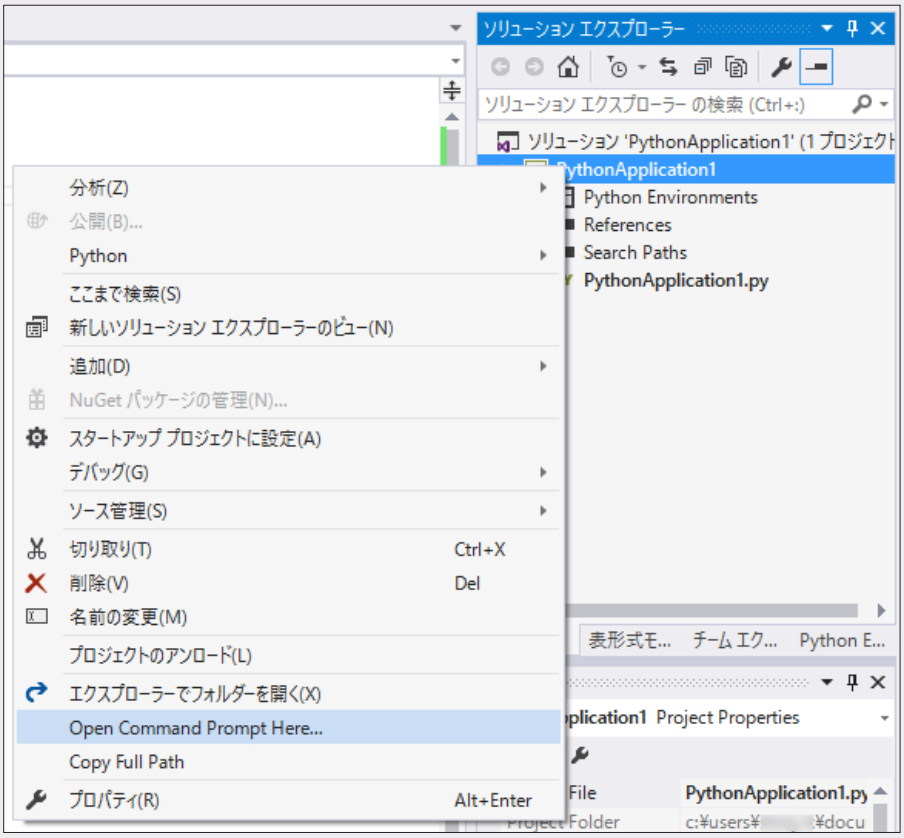
VS からこのモジュールを実行するには幾つかの方法がある。まず、ツールバーにある［開始］ボタンをクリックする方法だ。あるいはソリューションエクスプローラーからコマンドプロンプトを開いて、コマンドラインで python コマンドを利用する方法もある。また、［Interactive］ウィンドウでこのモジュールをインポートする方法もある。

最初の方法の［開始］ボタンをクリックすると以下のようにコマンドプロンプトが開く。

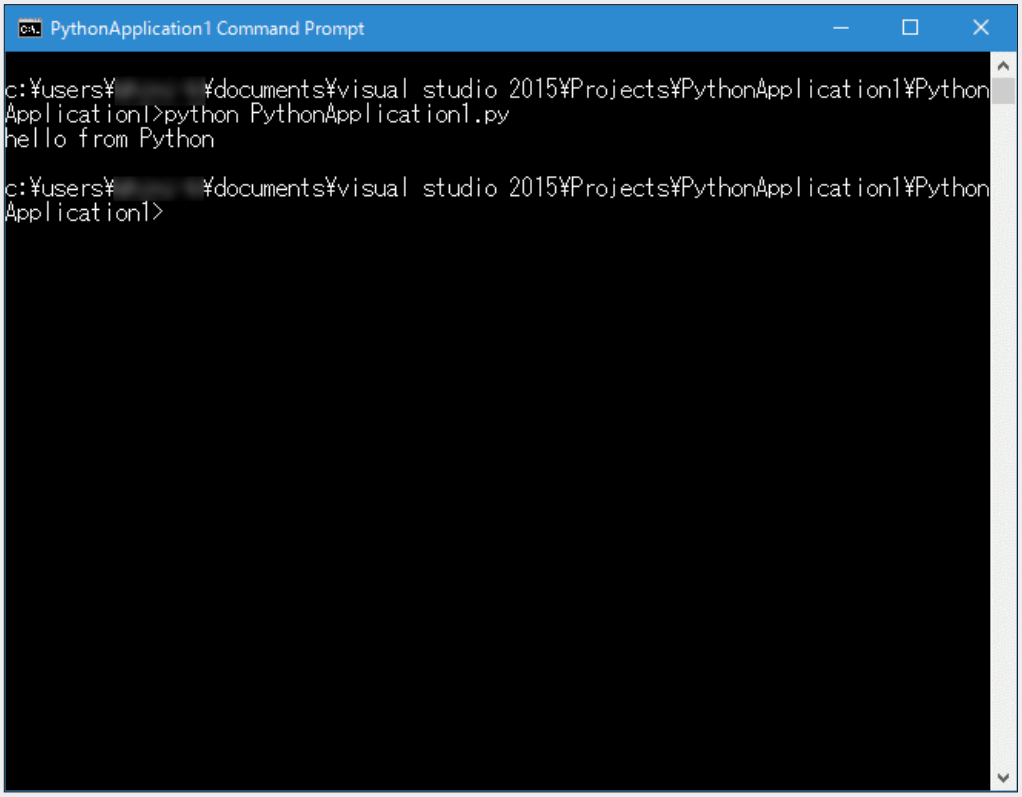


Python スクリプト（モジュール）の実行

先頭の print 関数により画面に出力が行われている。def 文による関数 hello の定義ももちろん実行されている。ただ、関数定義はされているが、使用されていないだけだ。ソリューションエクスプローラーからコマンドプロンプトを開いて、そこから python コマンドを実行した場合も同様だ。



プロジェクトを右クリックしてコンテキストメニューから［Open Command Prompt Here］（または［ここでコマンド プロンプトを開く］）を選択



コマンドプロンプトから python コマンドを実行

コマンドプロンプトを開いて python コマンドを実行

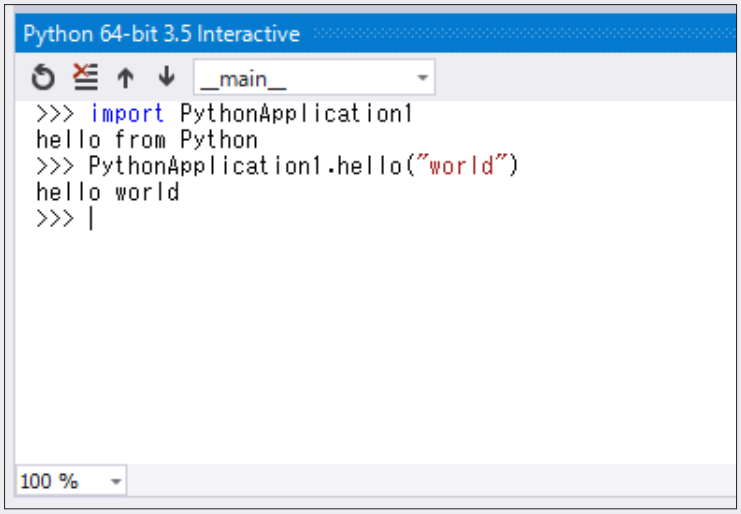
→目次に戻る

最後に［Interactive］ウィンドウで、今作成したモジュールをインポートしてみよう。これには以下のコードを［Interactive］ウィンドウに入力する。

```
import PythonApplication1
PythonApplication1.hello("world")
```

［Interactive］ウィンドウでの実行する Python コード

実行結果は次のようになる。



import 文で PythonApplication1 モジュールをインポートして、関数 hello を呼び出す

上に示した形式の import 文は最もシンプルな import 文の使い方だ。これにより PythonApplication1.py ファイルの内容が［Interactive］ウィンドウ上の Python セッションに PythonApplication1 モジュールとして取り込まれる（その際には Python のインタープリターにより組み込み関数 print が実行され、次に関数を定義している def 文も実行される）。注意点としては、この形式でモジュールをインポートした場合には、上で示したように関数 hello の呼び出しでモジュール名（ファイル名から拡張子を除いたもの）が必要になる点だ（つまり、モジュールは名前空間の役割を果たす）。

import 文

今見たように import 文は何らかのコンテキストにモジュールを取り込む（上の場合は［Interactive］ウィンドウのセッションにモジュールをインポートしたが、あるファイルで別のファイルの内容をインポートすることも可能だ）。上では一番シンプルな import 文を使ってみたが、他にも構文がある。以下に import 文の構文を示す。

```
# シンプルな形式
import モジュール名

# モジュールに別名を付けてインポート
import モジュール名 as 別名

# モジュールから特定の要素をインポート
from モジュール名 import インポートするもの

# 特定の要素に別名を付けてインポート
from モジュール名 import インポートするもの as 別名

# モジュールから全てをインポート
from モジュール名 import *
```

import 文の構文

一番上のシンプルな形式は既に見た。その次の形式は、インポートするモジュールに別名を付けるものだ。以下に［Interactive］ウィンドウでの実行例を示す。

```
>>> import PythonApplication1 as PA1
>>> PA1.hello("hoge")
hello hoge
```

モジュールに別名を付ける

モジュール名が非常に長い場合や、使いたいモジュールの名前が重複している場合などに、別名を与えることでアクセスを簡単にしたり、重複を避けたりできる。

次の形式はモジュールから選択的にインポートする。例えば、以下のように PythonApplication1 モジュールに関数を追加したとする。

```
print("hello from Python");

def hello(name):
  print("hello", name)

def goodbye(name):
  print("goodbye", name)
```

関数 goodbye を追加

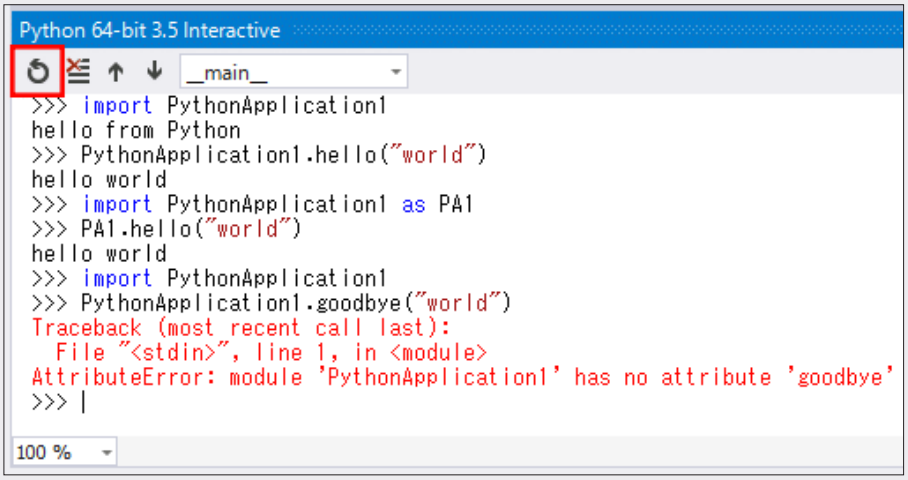
このモジュールから関数 goodbye だけをインポートするには次のようにする。

```
from PythonApplication1 import goodbye
goodbye("world")
```

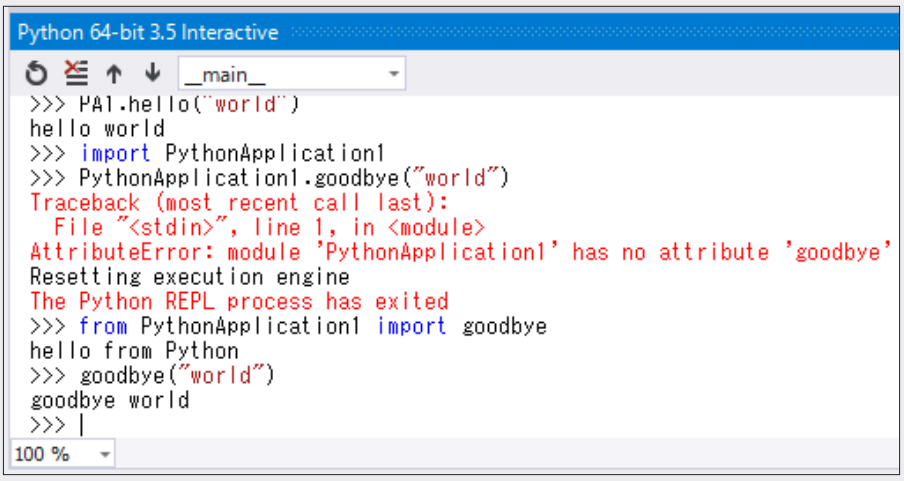
PythonApplication1 モジュールから関数 goodbye のみをインポート

→目次に戻る

なお、あるモジュールを既にインポート済みの Python セッション（［Interactive］ウィンドウや python ／ idle コマンドで起動される対話環境）で、修正後のモジュールをセッションに反映させるには importlib モジュールの関数 reload でモジュールを再読み込みする必要がある（以下のコラムを参照）。［Interactive］ウィンドウではウィンドウ左上の［リセット］ボタンをクリックしてセッションの内容をリセットしてから、import 文でモジュールを再度インポートしてもよい。ここでは後者の方法を使って、PythonApplication1 モジュールを読み込み直しながら、関数 goodbye だけをインポートしてみよう。



この状況では PythonApplication1.goodbye 関数は呼び出せていない。［Interactive］ウィンドウ左上の［リセット］ボタン（赤枠内）をクリックする



対話環境をリセットしてからモジュールを再インポートしたので、修正後のモジュールの内容が反映された

Interactive］ウィンドウ上のセッションをリセットして、モジュールをインポートし直す

最初の画面の上の方では、これまでに見てきた操作（PythonApplication1 モジュールのインポート、別名の付与）が行われている。画面だけでは分かりづらいのだが、その後で PythonApplication1.py ファイルの内容を修正している。その後、import 文で再インポートをしたのだが、関数 goodbye の呼び出しに失敗している（上の画像の最初の import 文では「hello from Python」とメッセージが表示されているが、それ以外の import

文ではこのメッセージが表示されていない。これは修正内容が実行されていない=セッションに反映されていないということだ)。

ウィンドウ左上の［リセット］ボタン（赤枠内）をクリックした後の操作が次の画面だ。「Resetting execution engine」「The Python REPL process has exited」という 2 行が表示され、環境がリセットされたことが示されたところで、「from モジュール名 import インポートするもの」形式の import 文を実行している。

ここで重要なのは、この形式でインポートしたものは、ローカルスコープにその名前が導入されるので、アクセスするのにモジュール名を必要としない点だ。そのため、「goodbye("world")」のように関数名を直接記述できる。

importlib.reload 関数でモジュールの修正内容を対話環境に反映する

上で見たように、VS の［Interactive］ウィンドウでは［リセット］ボタンを使えば、対話環境をリセットして、モジュールをインポートし直すといったことが簡単に行える。しかし、python ／ idle コマンドで起動した対話環境ではこの方法は使えないし、［Interactive］ウィンドウであっても多数のモジュールをインポートして動作を試しているといった場合には、リセットするたびに必要なモジュールを全てインポートし直さなければならない。そうしたときには importlib モジュールの関数 reload を使うのが便利だ。以下に使用例を示す。

```
>>> import importlib  # importlibモジュールをインポート
>>> importlib.reload(PythonApplication1)  # 関数reloadを呼び出す
hello from Python
<module 'PythonApplication1' from '.¥¥PythonApplication1.py'>
>>> PythonApplication1.goodbye("world")  # 修正後の内容が反映された
goodbye world
```

モジュールの内容を再読み込み

関数 reload を呼び出す際には引数にモジュール名を指定するが、これは既に現在のセッションに導入済みの名前なのでシングルクオート／ダブルクオートで囲む必要がない点には注意しよう。

4 番目の形式では選択的なインポートをした上で、それに別名を付ける。以下に［Interactive］ウィンドウでの実行例を示す。

```
>>> from PythonApplication1 import hello as foo
>>> foo("world")
hello world
>>> hello("world")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'hello' is not defined
```

インポートした関数に別名を付ける

→目次に戻る

3 番目（「from モジュール名 import インポートするもの」）の形式と同様、インポートした名前はローカルスコープに導入されるのでモジュール名なしで、それを直接利用できる。また、元の名前では利用できない（最後のエラーに注目）。

最後の形式（「from モジュール名 import *」）はモジュール内の全ての要素をローカルスコープに導入する。以下に例を示す（ここでは［リセット］ボタンでセッションをリセットしたものとする）。

```
>>> from PythonApplication1 import *
hello from Python
>>> hello("world")
hello world
>>> goodbye("world")
goodbye world
```

全てをローカルスコープにインポートする

この形式であるモジュールから全てをインポートすることはあまり推奨されていない。これは未知のモジュールから全てをインポートすることで、既に定義されている（何らかの値や関数などが代入されている）名前がインポートしたもので上書きされる可能性があるからだ。

また、この形式では「全て」がインポートされると述べたが、実際には「_」で始まる名前のものはインポートされない。前章でも触れたが「_」で始まる名前は内部使用を意味するものであり、Python ではそのような名前が特別扱いされている。興味のある方は実際に試してみてほしい。

ここまでは Python のモジュールの基本中の基本だが、まだ重要なこともある。

その他の重要なこと

本項の最後にモジュールの基礎をマスターする上で知っておくべきことを幾つか取り上げておこう。

グローバル変数「__name__」

グローバル変数「__name__」には、あるモジュールがインポートされた場合にはそのモジュール名が、そうではなく「python PythonApplication1.py」のように実行された場合には「__main__」という値が設定される。

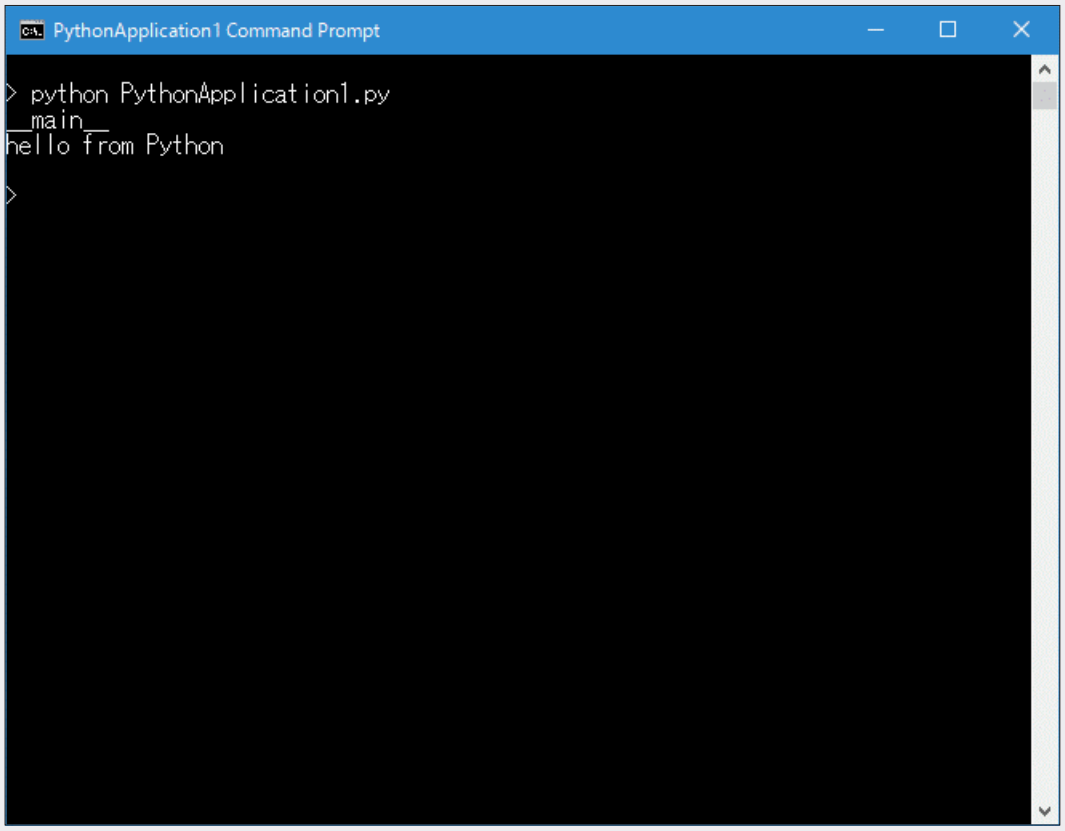
そして、この値を調べることで、モジュールが Python のスクリプトファイルとして実行されているのか、［Interactive］ウィンドウなどの対話環境や他の Python スクリプトでそのモジュールがインポートされている（つまり、ライブラリとして利用されている）のかを判断でき、それに応じて Python インタープリターによってモジュールが解釈／実行されたときの挙動を変更できる。

PythonApplication1.py ファイルを以下のように書き換えてみよう。

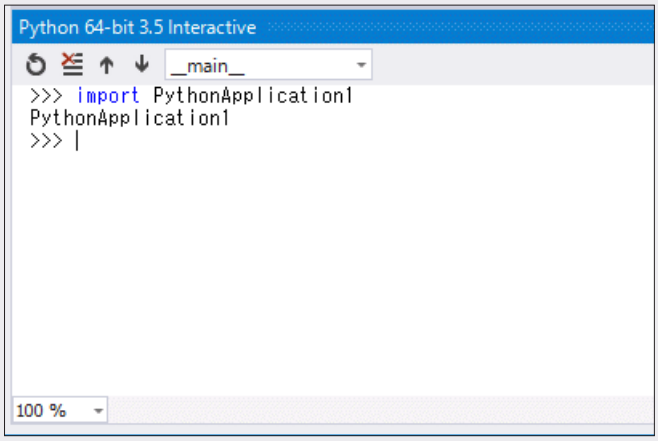
```
print(__name__)

def hello(name):
  print("hello", name)

def goodbye(name):
  print("goodbye", name)

if __name__ == '__main__':
  print("hello from Python");
```

__name__ の値に応じて挙動を変えるモジュール

これはスクリプトファイルとして実行されている場合には「hello from Python」とメッセージを表示するが、インポートされた場合にはメッセージを表示しないようにしたものだ（加えて、テスト用にスクリプトの冒頭で変数 __name__ の値を表示している）。次ページにコマンドラインから python コマンドで実行したときの結果と、［Interactive］ウィンドウでインポートしたときの結果を示す。



コマンドラインからの実行結果



[Interactive] ウィンドウでの実行結果

実行結果

1 つ目の画面がコマンドラインからスクリプトとして実行したものだ。変数 __name__ の値が「__main__」となっていて、if 節のブロックの内容が実行されていることが分かる。2 つ目の画面は［Interactive］ウィンドウでインポートをした場合のものだ。こちらではファイル名（から拡張子を除いたもの）が変数 __name__ の値になっていて（モジュール名）、if 節のブロックは実行されていない。

このようにすると、以下で説明するコマンドライン引数と組み合わせて、モジュールを単独のプログラムとして実行した場合には、関数定義などを行った後に（C# でいうところの static void Main メソッドを呼び出すのと似

た感じで）ユーザーから受け取った引数を処理してその結果をユーザーに返送するようにする一方で、ライブラリモジュールとしてインポートされた場合には、余計な処理を行わないようにできる。

言葉で説明しても分かりにくいので、以下の「コマンドライン引数」項で実際の例を紹介する。

コマンドライン引数

Python でコマンドライン引数を処理するには、sys モジュールが提供するリスト argv を使用する。まずは簡単な例を示す。ここでは PythonApplication1.py ファイルの内容を以下のように書き換えている。

```
from sys import argv  # sys.argvはコマンドライン引数を格納するリスト

if __name__ == '__main__':
  print(argv)

print("hello from Python module")
```

与えられた引数をそのままエコーバックする

実行結果を以下に示す。

```
> python PythonApplication1.py arg1 arg2
['PythonApplication1.py', 'arg1', 'arg2']
hello from Python module
```

argv[0] には実行されたモジュールのファイル名が含まれる

結果を見ると分かる通り、argv[0] には実行されるモジュール（スクリプトファイル）の名前が格納される。そのため、「モジュールに渡す引数の数＋ 1」個の要素が sys.argv には渡されることになる。実際の第 1 引数には argv[1]、第 2 引数には argv[2] のようにしてアクセスできる。

もう 1 つ例を示す（ここでも PythonApplication1.py ファイルの内容を丸ごと書き換えている）。

```
from sys import argv
from re import fullmatch  # reモジュールから関数fullmatchをインポート

def add(arg1, arg2):
  if fullmatch('[-+]{0,1}¥d+', str(arg1)) and fullmatch('[-+]{0,1}¥d+', str(arg2)):
    return "sum: " + str(int(arg1) + int(arg2))
  else:
    return "concatnate: " + str(arg1) + str(arg2)

if __name__ == '__main__':
  if len(argv) < 3:
    print("usage: python PythonApplication.py arg1 arg2")
  else:
    print(add(argv[1], argv[2]))
```

引数が整数値なら加算、そうでなければ文字列として連結

ここでは上で見た sys モジュールのリスト argv に加えて、re モジュール（正規表現モジュール）の関数 fullmatch をインポートして、関数 add を定義している。関数 add 内では re モジュールの関数 fullmatch を使って渡された 2 つの値がどちらも整数として扱えるかどうかを判定して、そうであればその和を、そうでなければ文字列としてそれらを連結している。さまざまな箇所で組み込み関数の int と str を使って、渡された値を整数化したり文字列化したりしているのは、関数 add に実際に渡される値の型が特定できないからだ（例としてあまりよくなかった……）。

そして、スクリプトとして実行されたときにはコマンドライン引数の数を調べて、数が足りなければ、使い方を示すメッセージを表示し、そうでなければコマンドライン引数を渡して関数 add を呼び出すようにしている。これ以上の詳細な説明は不要だろう。正規表現についてはレファレンスマニュアルの「6.2. re」を参照してほしい。

これをスクリプトとして実行した結果を以下に示す。

```
> python PythonApplication1.py 1 -1
sum: 0

> python PythonApplication1.py "hello " world
concatnate: hello world

> python PythonApplication1.py
usage: python PythonApplication.py arg1 arg2
```

スクリプトとして実行

一方、［Interactive］ウィンドウでインポートした場合の結果を以下に示す。

```
>>> from PythonApplication1 import add
>>> add("100", "200")
'sum: 300'
>>> add("hello ", "world")
'concatnate: hello world'
>>> add(1, 2)
'sum: 3'
```

モジュールとして対話環境にインポート

今見たようにグローバル変数 __name__ の値をチェックして、それに応じて処理を切り分けることで、モジュールを独立したスクリプトファイルとしても、ライブラリのようにも使えるようになる。

◇ ◇ ◇ ◇ ◇ ◇

本章では Python のモジュールの基本を紹介した。次章では複数のモジュールを組織化して扱うための機構である「パッケージ」と、モジュールに関して積み残した事項を紹介する。

特集：Visual Studio で始める Python プログラミング

8.Python のパッケージの基本も押さえる

本章では複数のモジュールを組織的に取り扱うための機構であるパッケージについて見ていこう。

前章では Python のモジュールの基本を取り上げた。モジュールは Python コードの再利用性を高める仕組みだが、プロジェクトが大規模になってくると、より高レベルな再利用機構が必要になる。本章では、複数のモジュールを組織化して取り扱うために使える Python のパッケージについて見ていこう。

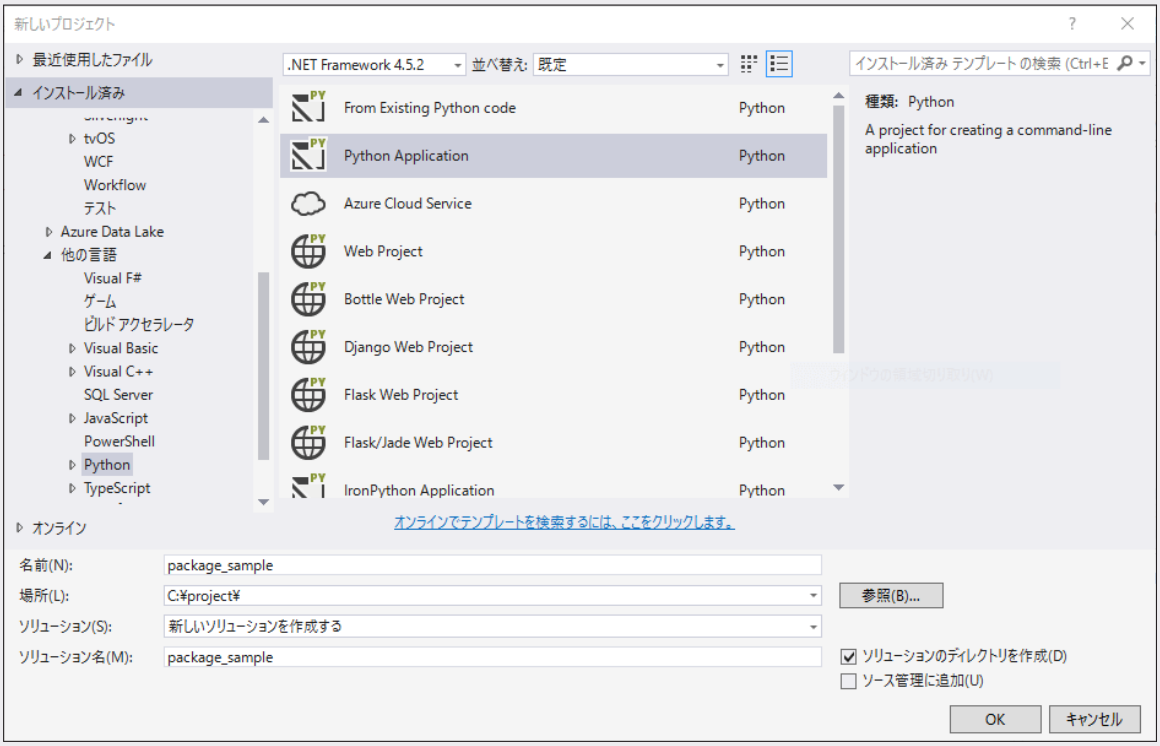
パッケージとは

前章でも述べたように、Python のモジュールとは単独のファイルに記述された Python のプログラム群（関数やクラス、変数など）のことだ。モジュールに記述されたクラスや関数、変数などは他の Python モジュール（Python スクリプト）で「インポート」して利用できる。シンプルなレベルでコードの再利用を可能にするのがモジュールである。

これに対して、パッケージは複数のモジュールで構成される、より大規模な再利用可能コード群だといえる。Python では多くの場合、何らかのディレクトリ以下に含まれるサブディレクトリ（ある場合）およびモジュールによってパッケージが構成される。

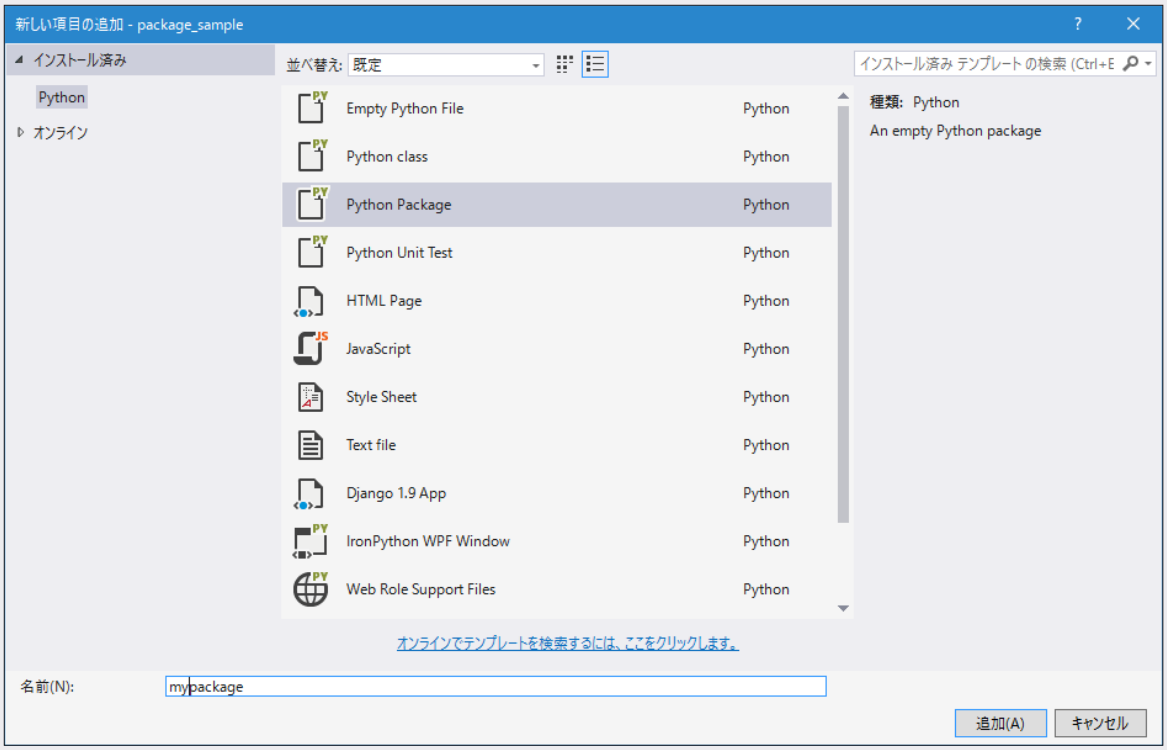
パッケージの作成

では、実際に VS でパッケージを作成してみよう。なお、以下では新規に「package_sample」という名前の Python プロジェクトを作成した。



新規に「package_sample」プロジェクトを作成

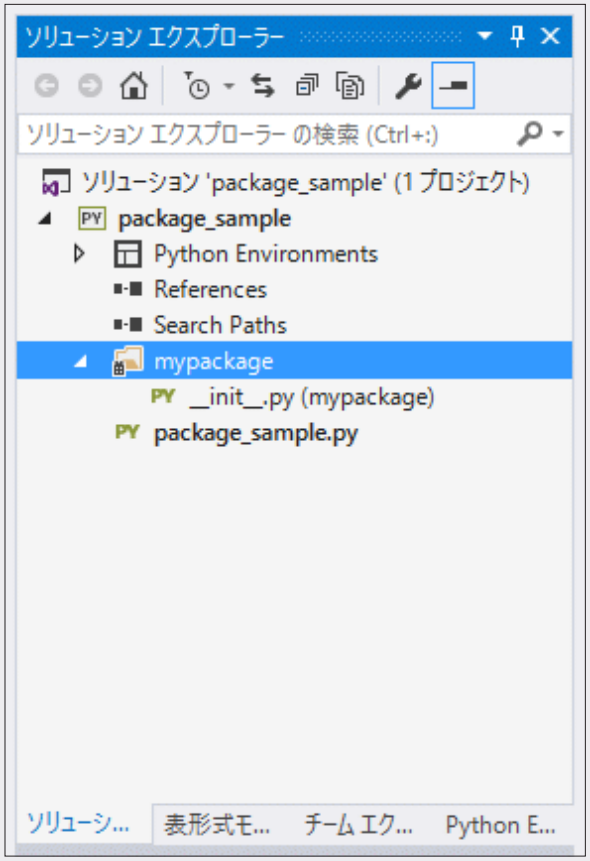
PTVS ではパッケージの作成も簡単に行える。ソリューションエクスプローラーでプロジェクトを右クリックして、コンテキストメニューから［追加］－［新しい項目］を選択したら、［新しい項目の追加］ダイアログで［Python Package］（または［Python パッケージ］）を選択するだけだ。ここではパッケージ名として「mypackage」を指定している。



プロジェクトへのパッケージの追加

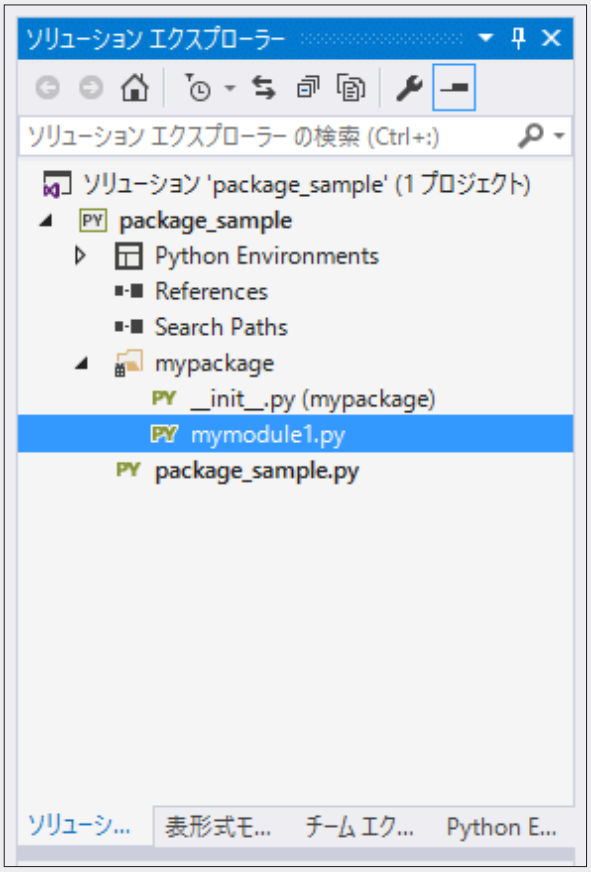
→目次に戻る

これにより、プロジェクトにパッケージ名と同じ名前のディレクトリが追加され、その下に「__init__.py」ファイルが作成される。なお、ここでは VS が提供する GUI を利用してパッケージを作成したが、その実体は下の画像のようにディレクトリと __init__.py ファイルなので、コマンドプロンプトなどから手作業で同じことをしても構わない。



パッケージ追加後のプロジェクト構成

通常のパッケージでは、パッケージを構成するディレクトリには上で見た __init__.py ファイルが必要となる。特に設定が必要なければ、このファイルの内容は空でもよいが、パッケージの読み込み時に何らかの初期化処理を行うようなコードを記述することもできる。これについては後で見るとして、今作成した mypackage パッケージにモジュール（Python のコードを含むファイル）を追加しておこう。VS でモジュールを追加するには先と同様な手順を行い、［新しい項目の追加］ダイアログで［Empty Python File］（または［空の Python ファイル］）を選択すればよい（画像は省略）。ここでは「mymodule1.py」ファイルを追加する。追加のプロジェクト構成は次のようになる。



モジュール追加後のプロジェクト構成

モジュールを追加したら、「mymodule1.py」ファイルにちょっとしたコードを記述しておく。ここでは次のコードを書いてみる。

```
print("hello from mypackage.mymodule1")

def foo():
  print("hello from foo on mypackage.mymodule1")
```

テスト用のコード（mypackage¥mymodule1.py ファイル）

簡単なコードだが、パッケージの振る舞いを調べるにはこれで十分だ。

パッケージ／モジュールのインポート

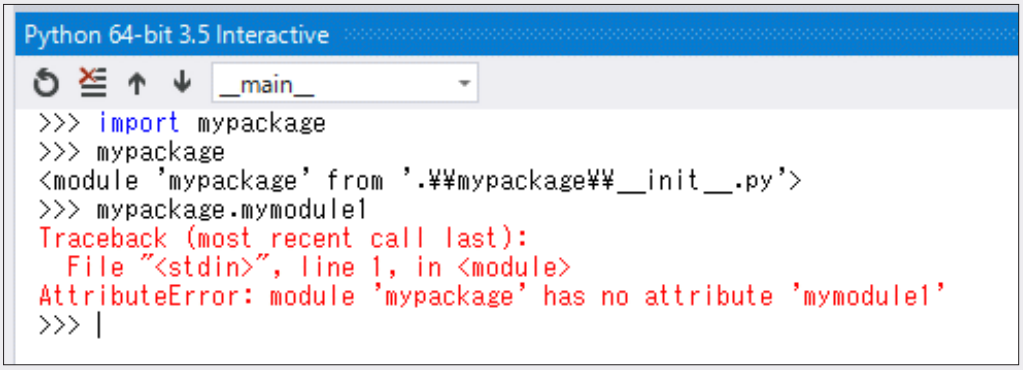
コードを書いたら、次にパッケージをインポートしてみよう。これには前章で紹介した import 文を使用できる。ここでは次のようにして、［Interactive］ウィンドウでパッケージをインポートしてみる。

```
import mypackage
mypackage
mypackage.mymodule1
```

パッケージとモジュールのインポート

［Interactive］ウィンドウで 2 行目と 3 行目を実行すると、mypackage パッケージおよび mypackage パッケージに含まれる mymodule1 モジュールについての簡単な情報が表示される（と予想される）。なお、ここで見たようにパッケージに含まれるモジュールは「パッケージ名 . モジュール名」のようにパッケージ名とモジュール名を「.」で区切って参照する。モジュールがそこに含まれる要素を束ねる名前空間のように機能するのと同様に、パッケージはそこに含まれるモジュールを束ねる名前空間のように機能するということだ。

［Interactive］ウィンドウで上のコードを実際に試した例を以下に示す。



トップレベルのパッケージをインポートしたからといって、その下のモジュールが自動的にインポートされるわけではない

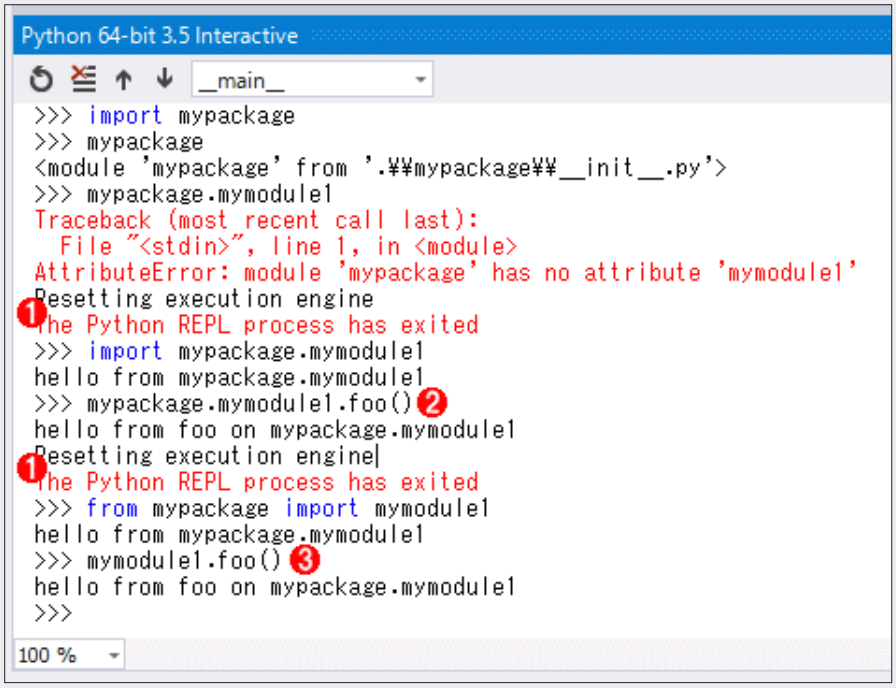
予想とは異なり、そのままの状態だとパッケージをインポートしただけでは、その下にあるモジュールまでは自動的にインポートされない（上の画像の最後のエラーを参照）。パッケージに含まれているモジュールをインポートするには幾つかの方法がある。以下に例を示す。

```
# 形式1
import mypackage.mymodule1

# 形式2
from mypackage import mymodule1
```

パッケージに含まれるをインポートする方法

形式 1 の場合、インポートしたモジュールには「パッケージ名 . モジュール名」としてアクセスする。形式 2 の場合はパッケージ名を省略して、モジュール名のみでアクセス可能だ（この辺は前章で見たモジュールのインポートと、その要素へのアクセスと同様と考えられる）。パッケージがさらにパッケージを含んでいるような場合には「import pkg1.pkg2.module1」あるいは「from pkg1.pkg2 import module1」のような形でインポートを行える。

以下に実行例を示す。なお、パッケージ／モジュールの読み込みを試してみる際には、パッケージがインポート済みの状態ではないように前回も見た［Interactive］ウィンドウ左上にある［リセット］ボタンをその都度クリックして、Python の実行環境を真っさらなものにしておくようにしよう（以下の画像の中では①で示した「Resetting execution engine」行と「The Python REPL process has exited」行が［リセット］ボタンによる対話環境リセットの出力）。コマンドプロンプトなどから Python の対話環境を起動している場合は、試すごとに対話環境を終了して（quit() コマンド）、環境を再起動するとよい。



パッケージに含まれるモジュールのインポートとモジュールに含まれる関数 foo の呼び出し

mymodule1 モジュールに含まれている関数 foo の呼び出しを見てみよう。②は形式 1 でインポートした場合の関数 foo の呼び出しだ。この場合は「パッケージ名 . モジュール名 . 関数名」でアクセスを行っている。②は形式 2 でインポートした場合のもの。こちらでは「モジュール名 . 関数名」でアクセスをする。これは前者ではローカルスコープに「mypackage」という名前（識別子）が導入され、後者では「mymodule1」という名前が導入されているからだ。

現在のローカルスコープ（現在実行中の Python コードが直接アクセス可能な最も内側のスコープのこと。スコープと名前空間については次章で取り上げることにする）に導入されている名前を調べるには組み込み関数 dir を使用できる。以下に例を示す。

```
# 上の形式1でモジュールをインポートして現在のローカルスコープを参照
import mypackage.mymodule1
dir()
dir(mypackage)

# Pythonの対話環境をリセットする

# 上の形式2でモジュールをインポートして現在のローカルスコープを参照
from mypackage import mymodule1
dir()
dir(mymodule1)
```

組み込み関数 dir を使用してローカルスコープを調べる

形式 1 でインポートした場合の結果は次のようになる。

```
>>> import mypackage.mymodule1
hello from mypackage.mymodule1  # mymodule1モジュールが読み込まれたことを示すメッセージ
>>> dir()  # ローカルスコープに含まれている名前を表示
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'mypackage']
>>> dir(mypackage)  # 「mypackage」に含まれている名前を表示
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__path__', '__spec__', 'mymodule1']
```

形式 1 でインポートした場合の組み込み関数 dir の実行結果

ローカルスコープには「mypackage」が含まれていることと、「mypackage」には「mymodule1」が含まれていることが分かる。このため、この場合には「mypackage.mymodule1」のようにアクセスをすることになる。

一方、形式 2 でインポートした場合の結果は次のようになる。

```
>>> from mypackage import mymodule1
hello from mypackage.mymodule1
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'mymodule1']
>>> dir(mymodule1)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'foo']
```

形式 2 でインポートした場合の組み込み関数 dir の実行結果

この場合はローカルスコープに「mymodule1」が導入されていること分かる。

もちろん、以下のようにして関数 foo を直接インポートすることも可能だ。

```
from mypackage.mymodule1 import foo
foo()
```

関数 foo のインポート

次に先ほど簡単に説明をした __init__.py ファイルについて見ていこう。

__init__.py ファイルの __all__ 変数

ところで、前章ではインポートの形式として「from モジュール名 import *」を取り上げた。これはモジュールに含まれる全ての要素をローカルスコープに導入するものだ。この形式のインポートはパッケージについても以下のようにして同様に使える。ここで email パッケージは Python に標準で組み込まれているパッケージであり、電子メール関連の処理を行う複数のモジュールで構成されている。前回も述べたように、モジュールやパッケージに含まれているものを全て無鉄砲にインポートするのはよろしくないが、ここでは例としてこれを行っている。

```
from email import *
dir()
```

email モジュールのインポート

［Interactive］ウィンドウでの実行結果は次のようになる（改行は適宜挿入したもの）。

```
>>> from email import *
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__',
'base64mime', 'charset', 'encoders', 'errors', 'feedparser', 'generator',
'header', 'iterators', 'message', 'message_from_binary_file',
'message_from_bytes', 'message_from_file', 'message_from_string', 'mime',
'parser', 'quoprimime', 'utils']
```

email パッケージから全てをインポート

これと同様なことを先ほどの mypackage について行うとどうなるだろうか。これを（［Interactive］ウィンドウの環境をリセットしてから）試した結果が以下だ。

```
>>> from mypackage import *
>>> dir()  # mypackageが表示されない
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']
>>> mypackage  # mypackageという名前は定義されていない
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'mypackage' is not defined
```

自作パッケージで同じことを試してみると……

Python のチュートリアルドキュメントの「6.4.1. パッケージから * を import する」には「パッケージの __init__.py コードに __all__ という名前のリストが定義されていれば、from package import * が現れたときに import すべきモジュール名のリストとして使う」「もしも __all__ が定義されていなければ、実行文 from sound.effects import * は、パッケージ sound.effects の全てのサブモジュールを現在の名前空間の中へ import しません」などと書いてある。

要するに、「from パッケージ import *」を実行したときに、パッケージ中のどのモジュールをインポートさせるかはパッケージの作者が明示的に指定をしなければならず、そのためには __init__.py ファイルの中で __all__ 変数にインポートするものを指定する必要があるということだ。

そこで、今まで空だった __init__.py ファイルを編集してみることにしよう。といっても、ここでは次の 1 行を追加するだけだ。

```
__all__ = ["mymodule1"]
```

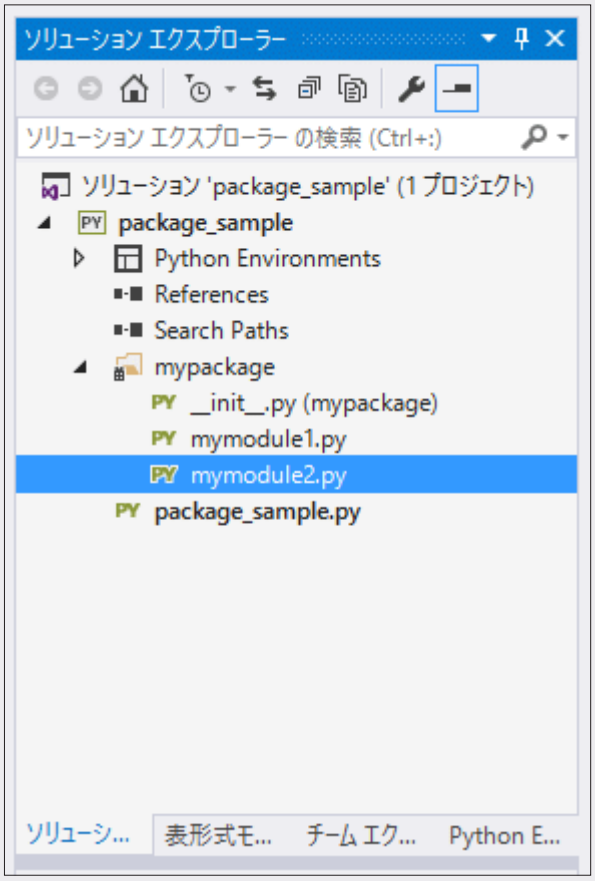
「from mypackage import *」でインポートされるものを指定（mypackage¥__init__.py ファイル）

__all__ 変数にはインポートさせたいモジュールを要素としたリストを代入する。以上のコードを書いてから、先ほどと同じことをしてみると次のようになる。

```
>>> from mypackage import *
hello from mypackage.mymodule1
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__',
'mymodule1']
```

mymodule1 がローカルスコープに導入された

ここでパッケージに mymodule2.py ファイルを追加してみよう。



mypackage パッケージに mymodule2.py ファイルを追加

mymodule2.py ファイルには mymodule1.py ファイルと同様にインポート時にメッセージを表示し、同じくメッセージを表示するだけの関数 foo を定義してある。

```
print("hello from module2 in mypackage")

def foo():
  print("this is mypackage.mymodule2.foo speaking")
```

mypackage¥mymodule2.py ファイルの内容

この状況で（[Interactive] ウィンドウをリセットして）「from mypackage import *」を実行した結果を以下に示す。

```
>>> from mypackage import *
hello from mypackage.mymodule1  # 読み込まれたのはmymodule1のみ
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__',
'mymodule1']
>>> from mypackage import mymodule2  # mymodule2を明示的にインポート
hello from module2 in mypackage
>>> mymodule2.foo()
this is mypackage.mymodule2.foo speaking
```

__init__.py ファイルに記述していないモジュールは「from mypackage import *」ではインポートされない

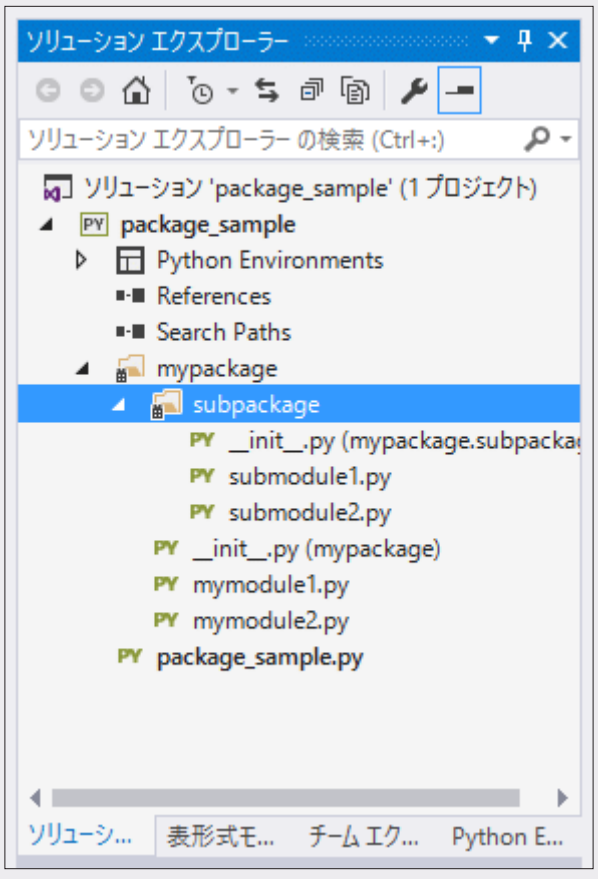
→目次に戻る

__init__.py ファイルの __all__ 変数では mymodule2 モジュールをインポートするような記述は行っていないので、このパッケージには mymodule1 ／ mymodule2 の 2 つのモジュールがあるにもかかわらず「from mypackage import *」で mymodule2 モジュールはインポートされない。このように __init__.py ファイルで設定を行っておけば、自分のパッケージを誰か別の人が使うときに安心して（あるいは開発者が意図した形で）「from パッケージ名 import *」を実行できるようになる。

本章の最後に、パッケージがパッケージを含んでいる場合についても簡単に見ておこう。

サブパッケージを含むパッケージ

ここでは以下のように mypackage パッケージ内に subpackage パッケージを作成し、そこに submodule1.py ファイルと submodule2.py ファイルを作成した。



サブパッケージとサブモジュールの作成

2 つの「.py」ファイルの内容はこれまでと同様なので、コードは割愛する。mypackage¥__init__.py ファイルは次のように変更した。

```
__all__ = ["mymodule1", "mymodule2"]
```

mypackage¥__init__.py ファイルの内容

mypackage¥subpackage¥__init__.py ファイルの内容は以下の通り。

```
__all__ = ["submodule1", "submodule2"]
```

mypackage¥subpackage¥__init__.py ファイルの内容

すでに述べたように、パッケージがサブパッケージを含んでいる場合には、以下のようにしてサブパッケージのインポートが可能だ（[Interactive] ウィンドウでの出力結果）。

```
>>> import mypackage.subpackage.submodule1
submodule 1
>>> mypackage.subpackage.submodule1.bar()
mypackage.subpackage.submodule1.bar
>>> from mypackage.subpackage import submodule2
hello from submodule 2
>>> submodule2.bar()
hello from bar in mypackage.subpackage.submodule2
```

サブパッケージ／サブモジュールのインポート

ただし、注意点が 1 つある。それは、先ほど見た「from パッケージ名 import *」ではサブパッケージの内容はインポートされないという点だ。実際に試した例を以下に示す。

```
>>> from mypackage import *
hello from mypackage.mymodule1
hello from module2 in mypackage
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__',
'mymodule1', 'mymodule2']
>>> from mypackage.subpackage import *
submodule 1
hello from submodule 2
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__',
'mymodule1', 'mymodule2', 'submodule1', 'submodule2']
```

「from パッケージ名 import *」ではサブパッケージはインポートされない

上の例で示しているように、サブパッケージに関しては別途インポートを行う必要がある。これは「import mypackage」としてトップレベルのパッケージをインポートしても、その下のモジュールが自動的にはインポートされなかったのと同様だ。

最後に、パッケージ内のモジュールから、同一パッケージ内の別モジュールを参照する方法について触れておこう。これにも幾つかの方法がある。一番簡単なのは以下のようにパッケージの階層をフルに記述することだ。

```
from mypackage.subpackage.submodule2 import bar as baz

print("submodule 1")

def bar():
  print("mypackage.subpackage.submodule1.bar")
```

パッケージ階層をフルに記述してインポートを行う（mypackage¥subpackage¥submodule1.py ファイル）

上のサンプルコードは「mypackage¥subpackage¥submodule1.py」ファイルのものであり、ここでは同じ階層（mypackage¥subpackage パッケージ）内の submodule2.py で定義されている関数 bar を「baz」という別名でインポートしている。

しかし、これは次のように「.」を利用して階層を逆順にさかのぼる相対的な方法（相対インポート）でも指定できる（「.」は現在のパッケージ、「..」は親パッケージといった具合）。

```
from .submodule2 import bar as baz

print("submodule 1")

def bar():
  print("mypackage.subpackage.submodule1.bar")
```

同一パッケージ内の別ファイルを相対インポート（mypackage¥subpackage¥submodule1.py ファイル）

「.」とパッケージ名／モジュール名の間には半角スペースを入れてもよいし、入れなくてもよい。さかのぼれるのはパッケージ内のみであることには注意しよう。以下にもう 1 つ例を示す。これは親階層（「..」により mypackage パッケージ階層を参照し、そこからさらに階層を下っている）。

```
from ..mymodule1 import foo
from .. subpackage.submodule2 import bar as baz

print("submodule 1")

def bar():
  print("mypackage.subpackage.submodule1.bar")
```

「..」により親階層を起点としてパッケージを参照

◇ ◇ ◇ ◇ ◇ ◇

本章では Python のパッケージの基礎知識を取り上げた。実際のところ、Python には「名前空間パッケージ」と呼ばれる「同じパッケージ名で始まる複数のパッケージを単一のパッケージに束ねる」機構もあるが、ここではこれは取り上げなかった。興味のある方は Python の言語レファレンス「5.2.2. 名前空間パッケージ」「PEP 420」（英語）などを参照してほしい。次章では、ここでも少し出てきたスコープ、名前空間について見ていこう。

特集：Visual Studio で始める Python プログラミング

9.Python における名前空間とスコープを理解する上でのポイントを押さえよう

簡単なコードを試しながら、Python の名前空間とスコープについての理解を深めよう。

前章では Python のパッケージを取り上げた。本章ではサンプルコードを実行しながら Python のスコープの仕組みと名前空間について考えてみる。

ちょっとテスト

まずは次のようなコードを対話環境で実行してみよう。ここでは VS の［Interactive］ウィンドウでも、コマンドプロンプトから起動した対話環境でも、Python に標準添付の統合環境「IDLE」でも何を使ってもよい。

```
l = list(range(1, 10))  # 組み込み関数listを使ってリストを作成
print(l)
list = 10               # 変数listに整数10を束縛
l = list(range(1, 10))  # これはエラーとなる
```

Python の挙動を試してみるコード（その 1）

実際に試してみると、結果は次のようになる。

```
>>> l = list(range(1, 10))
>>> print(l)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list = 10
>>> l = list(range(1, 10))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

最後の行でエラーが発生した

当たり前のことだが、変数 list に整数 10 を代入（束縛）したので、最後の行の「list(range(1, 10))」の呼び出しが失敗している（「TypeError: 'int' object is not callable」＝「整数オブジェクトは呼び出し可能ではない」）。見た目的には「組み込み関数 list」の定義を整数値で上書きしてしまったかのように見える。が、実はそうではない。続けて以下を試してみよう。del 文は指定された名前を削除するものだ。

```
del list  # 名前listを削除
l = list(range(1, 10))  # 組み込み関数listが再度呼び出し可能になる
print(l)
```

Python の挙動を試してみるコード（その 2）

実行結果を以下に示す。

```
>>> del list
>>> l = list(range(1, 10))
>>> print(l)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

組み込み関数 list が呼び出せるようになった

つまり、これは組み込み関数 list の定義が上書きされたのではなく、ローカルに定義された変数 list でそれが隠蔽（いんぺい）されたということだ。筆者もサンプルコードで「list = ……」とやってしまうことがあるので上のコードを例としたが、Python のスコープの仕組みにより、このようなことが起こるのだ。

大ざっぱにいうと、Python のスコープには次のような種類があると考えてよい。

1. ローカルスコープ
2. ローカルスコープを囲むスコープ（enclosing scope）
3. （モジュールレベルの）グローバルスコープ
4. 組み込み名前空間を表すスコープ

ローカルスコープとはもちろん現在実行しているコードを含む最小範囲のブロックのこと。ただし、気を付けたいのは、Python は C# などの言語ほど細かくはスコープを生成しない点だ。大まかには関数定義やクラス定義は新たなスコープを生成するが、if 文や for 文などではスコープは生成されない。そのため、例えば以下のようなコードは Python では正しいコードとなる。

```
def test(x):
  if x > 60:
    msg = "pass"
  else:
    msg = "fail"
  print(msg)  # if文を抜けてもmsgは生きている
```

Python ではこれは正しいコード

→目次に戻る

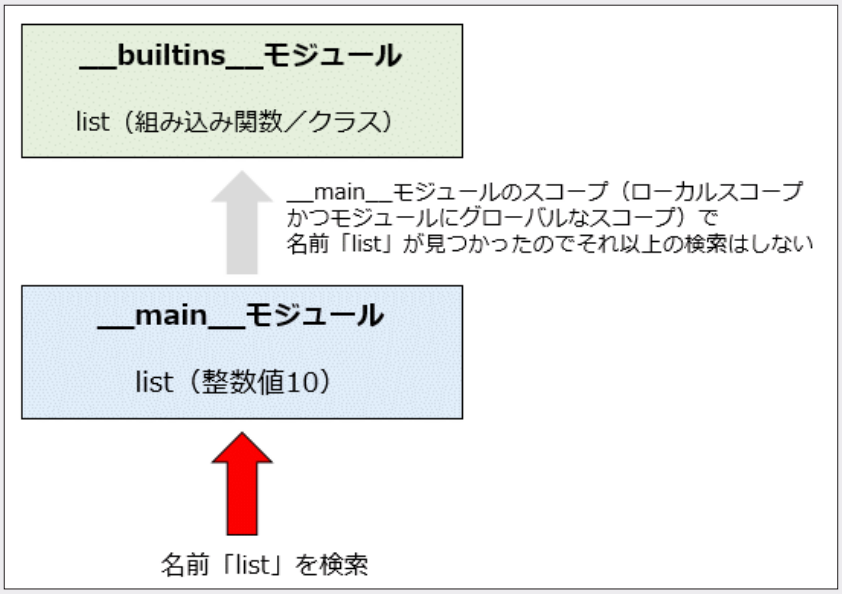
また、現在のローカルスコープを囲むスコープが存在している場合、それらは 2 の「ローカルスコープを囲むスコープ」（enclosing scope）として扱われる。典型的には関数定義内での関数定義がそうだ。この場合、外側の関数のスコープは内側の関数のスコープを囲むスコープとなる。

Python では単独のファイルが単独のモジュールとなることは以前にも説明した通りだが、モジュールのトップレベルで定義される名前はモジュールレベルでグローバルなスコープに含まれる（同時に、それらはモジュールのトップレベルでローカルな名前ともなる）。

最後の組み込み名前空間とは、Python に組み込みの関数やクラスなどを参照するために使われる名前空間のことだ（通常は変数 __builtins__ を利用して参照可能）。

Python で名前（整数、関数、クラスなどを参照するラベル=変数）を参照する際にはこれらのスコープが先に示した箇条書きの上から順に検索されていく。よって、ローカルスコープで定義されている名前があれば、それが参照されるし、グローバルなスコープまで検索しても名前が見つからなければ、組み込み名前空間から名前が検索され、そこにもなければ例外（NameError）が発生するということだ。

先ほどの例であれば、次の図のようになる。



Python のスコープと名前解決の順序

Python では対話環境で実行されるコードは「__main__」モジュールに含まれる。また、上の例では対話環境でそのまま変数 list を定義した（整数値 10 に名前 list を束縛した）ので、名前 list は __main__ モジュールのローカルスコープかつモジュール内でグローバルなスコープに存在している。そして、その後の「l = list(range(1, 10))」行ではローカルスコープに存在する名前 list が見つかるので、組み込み名前空間に含まれている名前 list は検索されないことになる。次の例で示した「del list」文によりローカルスコープに存在してい

た名前 list がなくなると、今度は組み込み名前空間に含まれる名前 list が参照されるようになる。というのが一連の流れだ。

何となく Python のスコープの仕組みが分かったところで、次に Python のスコープと名前空間についてもう少し深く探ってみよう。

名前空間とスコープとは

「Python チュートリアル」を見ると名前空間は「名前からオブジェクトへの対応付け（mapping）です。ほとんどの名前空間は、現状では Python の辞書として実装されています」とある。一方、スコープは「ある名前空間が直接アクセスできるような、Python プログラムのテキスト上の領域」となっている。

「ある名前空間が直接アクセスできるような」というのは「ある名前空間で定義されている要素に直接アクセス可能」といった意味合いだ。「直接アクセス可能」とは「名前空間名 . 要素名」ではなく「要素名」だけを指定してアクセスできるテキスト領域（Python のコードの一部分）を意味する（ここで「名前空間名」とは例えば、モジュール名= Python コードを含むファイルの名前やクラス名などのこと。モジュールやクラスはスコープと名前空間を新たに形成し、他の名前空間内の名前との衝突を避けるのに利用できる。この辺りの考え方は C# などの言語も Python も同様だ）。

これはつまり、「Python 言語リファレンス」で書かれている「ブロック内の名前の可視性を決めます」ということだ（こちらの方が、分かりやすい表現だろう）。

話をまとめると、Python では（多くの場合）名前空間は辞書として実装され、そこに名前とオブジェクトの対が保存される。そして、何らかの名前を検索する際にはローカルスコープから組み込み名前空間へと向かいながら、個々のスコープに対応する名前空間にその名前が存在しているかを検索していくことになる。

以下では、Python の組み込み関数を使いながら、この仕組みを少し探ってみよう。

組み込み関数 dir ／ globals ／ locals

Python ではスコープや名前空間に関する組み込み関数として以下のものがある。

組み込み関数	説明
dir	引数なしで呼び出すと現在のローカルスコープで定義されている（値に束縛されている）名前のリストを返す。引数に何らかのオブジェクトを渡すと、そのオブジェクトが持つ属性（メンバ）を返す
locals	現在のローカルスコープの名前空間の内容（辞書）を返す
globals	現在のグローバルスコープの名前空間の内容（辞書）を返す

組み込み関数 dir ／ locals ／ globals

組み込み関数 dir にはモジュールやクラスなどを渡すことで、そこに含まれるメンバを調べることも可能だ（コマンドプロンプトの dir コマンドが、ディレクトリの内容を表示するのと同様に、組み込み関数 dir は与えられたオブジェクトの内容を表示するものだと考えるとよい）。ただし、以下では組み込み関数 dir ではなく、次に述べる組み込み関数 locals と globals を使用して、Python のスコープや名前空間について調べていくことにしよう。

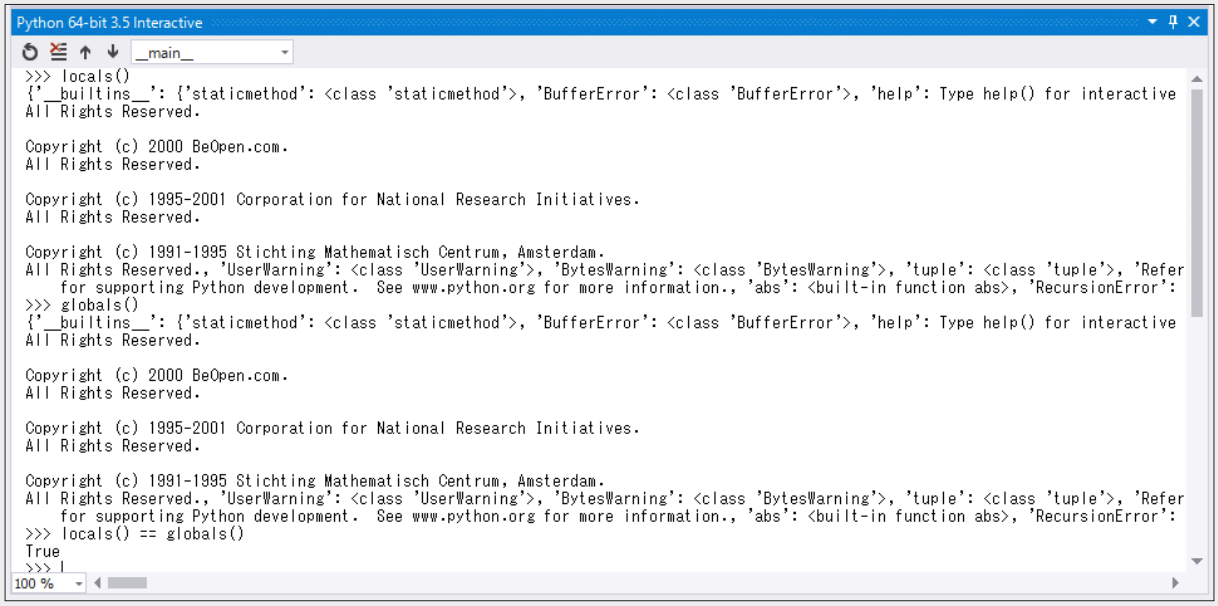
組み込み関数 locals はローカルスコープの名前空間の内容を表す辞書を、組み込み関数 globals はグローバルスコープの名前空間の内容を表す辞書を返す。

まずは［Interactive］ウィンドウで以下を試してみる。

```
locals()
globals()
locals() == globals()
```

ローカルな名前空間とグローバルな名前空間を表示

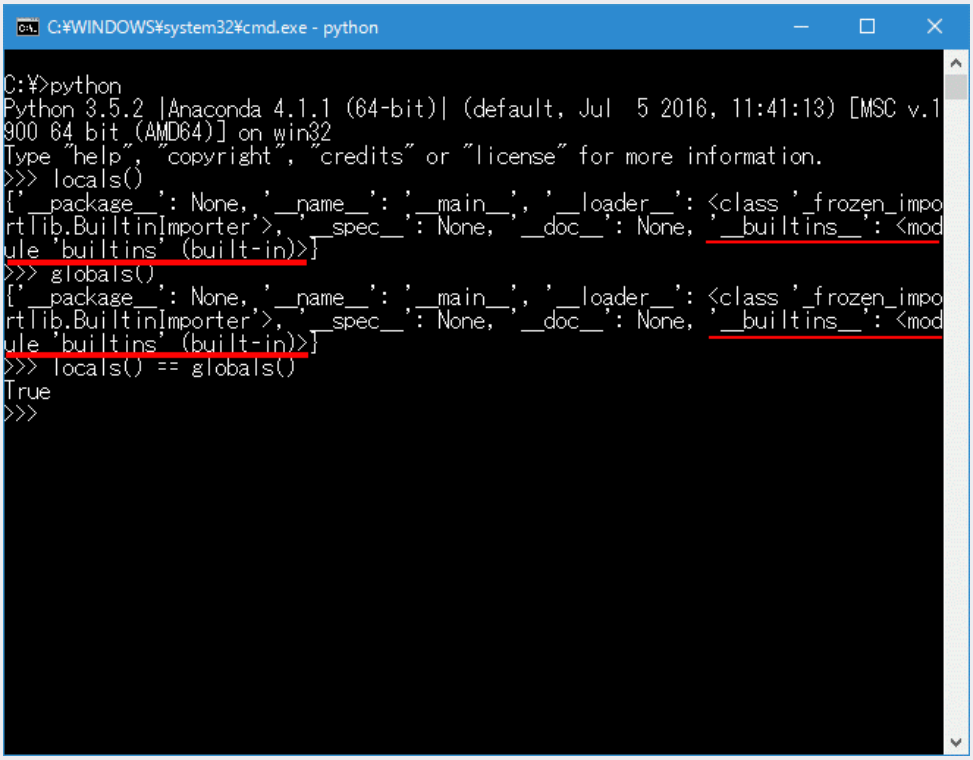
実行結果を以下に示す。



［Interactive］ウィンドウでの実行結果

何やらコピーライト表記などが表示されている。このコピーライト表記は組み込み名前空間（__builtins__ 変数が参照しているオブジェクト）で定義されている名前「copyright」の内容だ。VS の［Interactive］ウィンドウでは、__builtins__ 変数は組み込み名前空間を表す辞書となっていて、それが組み込み関数 locals ／ globals 呼び出しの結果、展開されて見えている。これは通常の Python の対話環境の動作とは異なっているので注意さ

れたい。コマンドプロンプトなどから python コマンドで対話環境を起動した場合の結果は次のようになる。



コマンドプロンプトから起動した Python の対話環境での実行結果

こちらでは __builtins__ 変数が参照しているのは組み込みモジュールの builtins であり、実行結果では「'__builtins__': <module 'builtins' (built-in)>」のように表示されている（上の画像の下線部）。このような動作の違いは VS でツールバーの［開始］ボタンをクリックして Python のスクリプトファイルを実行した場合でも同様であることから *1、以下では画面出力のシンプルさを重視して、コマンドプロンプトから python コマンドで対話環境を起動した場合の結果を基に話を進めていく。

*1 「Python 言語リファレンス」の「組み込みと制限付きの実行」を見ると、「__builtins__ は辞書かモジュールでなければなりません（後者の場合はモジュールの辞書が使われます）。デフォルトでは、__main__ モジュール中においては、__builtins__ は組み込みモジュール builtins です；それ以外の任意のモジュールにおいては、__builtins__ は builtins モジュール自身の辞書のエイリアスです」と書いてある。VS の［Interactive］ウィンドウでは通常は __main__ モジュールに含まれるコードを実行するので上の引用からすると「builtins モジュール」が見えそうなところだが、そうではなく「builtins モジュール自身の辞書のエイリアス」が見えている状態になっているということだ。

以上を踏まえて、次ページでは組み込み関数 locals ／ globals を使って Python のスコープや名前空間について調べていく。

また試してみよう

では、話を戻して、「ローカルな名前空間とグローバルな名前空間を表示」で示した 3 行のコードの実行結果を（コマンドプロンプトから起動した対話環境で）見てみよう。その結果は次のようになっていた（改行は筆者が適宜挿入している。以下同様）。

```
>>> locals()
{'__package__': None, '__name__': '__main__',
'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
'__doc__': None, '__builtins__': <module 'builtins' (built-in)>}
>>> globals()
{'__package__': None, '__name__': '__main__',
'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
'__doc__': None, '__builtins__': <module 'builtins' (built-in)>}
>>> locals() == globals()
True
```

実行結果

最後の「locals() == globals()」の結果が「True」となっていることから分かるように、対話環境のローカルスコープの名前空間の内容とグローバルスコープの名前空間の内容は最初は同一だ。先にも述べたが、対話環境は __main__ モジュールに含まれる形で実行されていて、そこで何らかの名前を定義した場合、それは __main__ モジュールでグローバルかつ __main__ モジュールにローカルな名前となる。試しに変数を 1 つ定義してみよう。

```
name = "insider.net"
locals()
globals()
```

変数を定義する

これを対話環境で実行すると、その結果は次のようになる。

```
>>> name = "insider.net"
>>> locals()
{'__package__': None, '__name__': '__main__', 'name': 'insider.net',
'__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__spec__': None, '__doc__': None,
'__builtins__': <module 'builtins' (built-in)>}
>>> globals()
{'__package__': None, '__name__': '__main__', 'name': 'insider.net',
'__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__spec__': None, '__doc__': None,
'__builtins__': <module 'builtins' (built-in)>}
```

変数がローカルの名前空間にもグローバルの名前空間にも現れる

今度は関数を定義してみよう。

```
def foo():
  bar = "bar"
  print("local namespace in foo:", locals())
  print("global namespace in foo:", globals())

foo()
print("local namespace in toplevel:", locals())
print("global namespace in toplevel:", globals())
```

関数内でローカル／グローバルな名前空間を調べる

実行結果は次のようになる。

```
>>> def foo():
...   bar = "bar"
...   print("local namespace in foo:", locals())
...   print("global namespace in foo:", globals())
...
>>> foo()
local namespace in foo: {'bar': 'bar'}
global namespace in foo: {'__package__': None, '__name__': '__main__',
'name': 'insider.net',
'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
'__doc__': None, 'foo': <function foo at 0x0000015133E57F28>,
'__builtins__': <module 'builtins' (built-in)>}
>>> print("local namespace in toplevel:", locals())
local namespace in toplevel: {'__package__': None, '__name__': '__main__',
'name': 'insider.net',
'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
'__doc__': None, 'foo': <function foo at 0x0000015133E57F28>,
'__builtins__': <module 'builtins' (built-in)>}
>>> print("global namespace in toplevel:", globals())
global namespace in toplevel: {'__package__': None, '__name__': '__main__',
'name': 'insider.net',
'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
'__doc__': None, 'foo': <function foo at 0x0000015133E57F28>,
'__builtins__': <module 'builtins' (built-in)>}
```

関数内部のローカルな名前空間にはそこで定義された名前だけが存在する

定義された関数はモジュールのグローバルスコープ／ローカルスコープの名前空間に登録されていることが分かる（強調書体で示した「'foo': <function foo at 0x0000015133E57F28>」という出力結果）。一方、関数内でローカルな変数は関数内部のローカルな名前空間にのみ存在していることも分かる（関数 foo 呼び出しの出力結果の先頭行）。

今度は先ほどトップレベルで定義した変数 name を関数 foo 内で使ってみよう。なお以下では、関数内部での組み込み関数 globals およびモジュールトップレベルでの組み込み関数 locals ／ globals の呼び出しは省略する（ので、［Interactive］ウィンドウでも表示結果は恐らく変わらないはずだ）。

```
def foo():
  bar = 'bar'
  print(name, bar)
  print(locals())

foo()
```

グローバルな変数を関数内で使用する

実行結果は次のようになる。

```
>>> def foo():
...   bar = 'bar'
...   print(name, bar)  # 変数nameとbarの内容を半角スペースで区切って表示
...   print(locals())
...
>>> foo()
insider.net bar
{'bar': 'bar'}
```

ローカルな名前空間には名前 name は存在しない

見ての通り、ローカルスコープの名前空間には名前 name は存在しないが、きちんと使用できている。これが先に述べたローカルスコープから始まる名前の可視化ということだ。名前がローカルスコープにないので、グローバルスコープに存在する名前が検索されて参照されている。

では、関数内でグローバルスコープにある変数（グローバル変数）と同名の変数に値を代入してみるとどうなるだろう。

```
def foo():
  name = "windows server insider"
  bar = "bar"
  print(locals())

foo()
print(name)
```

グローバル変数と同名の変数を定義

実行結果を以下に示す。

```
>>> def foo():
...   name = "windows server insider"
...   bar = "bar"
...   print(locals())
...
>>> foo()
{'name': 'windows server insider', 'bar': 'bar'}
>>> print(name)
insider.net
```

グローバル変数は隠蔽される

この場合には、グローバルスコープに存在する変数 name の値が書き換えられる（再束縛される）のではなく、ローカルスコープに新たに名前が導入される（なお、グローバル変数や「ローカルスコープを囲むスコープ」の変数を操作する方法については後述する）。

最後の例として「ローカルスコープを囲むスコープ」を作ってみよう。

```
def makeadder100and(x):
  y = 100
  print(locals())
  def adder(z):  # ネストした関数
    print(locals())
    return x + y + z  # ローカルスコープを囲むスコープ内の変数にアクセス
  return adder

add101 = makeadder100and(1)
add101(100)
```

ネストした関数からは外側のスコープにある関数にアクセスできる

関数 makeadder100and は、パラメーター x に値を受け取るとともに、その内部でローカル変数 y と関数 adder を定義している（戻り値は関数 adder）。ネストした関数 adder ではパラメーター z に受け取った値と、それを「囲むスコープ」に存在する変数 x と y にもアクセスをしている。そして、それぞれの関数の内部ではローカルな名前空間に存在する値を出力している。

実行結果は次のようになる。

```
>>> def makeadder100and(x):
...   y = 100
...   print(locals())
...   def adder(z):
...     print(locals())
...     return x + y + z
...   return adder
...
>>> add101 = makeadder100and(1)
{'y': 100, 'x': 1}  # 関数makeadder100andの名前空間の内容
>>> add101(100)
{'z': 100, 'x': 1, 'y': 100}  # 関数adderの名前空間の内容
201
```

ネストした関数のローカルな名前空間には外側のスコープに存在する変数 x と y も存在している

関数内からグローバル変数にアクセスしても、ローカルな名前空間にはそれが存在していなかったのとは異なり、今回はネストした関数 adder では外側のスコープの変数 x と y も名前空間内に存在している。このようにクロージャを形成するネストした関数では、それを囲むスコープ内の変数も名前空間に導入される（それらの変数を使用している場合）。このような変数のことを「自由変数」と呼ぶ（「Python 言語リファレンス」の「4.2.1. 名前の束縛」では「ある変数があるコードブロック内で使われていて、そのブロックで定義はされていないなら、それは自由変数（free variable）」となっている。この範疇にはグローバルな名前空間に存在する変数は含まれず、外側のスコープの名前空間に存在する変数が「自由変数」となるようだ）。

先ほど「グローバル変数やローカルスコープを囲むスコープの変数を操作する方法については後述する」と述べた。そのために用意されている global ／ nonlocal キーワードを本章の最後に紹介しておこう（global キーワードは「global 文」で使用されるキーワードであり、組み込み関数の「globals」とは異なることに注意）。

global ／ nonlocal キーワード

Python には global ／ nonlocal という他の言語ではあまり見られないキーワードがある。これらは現在のスコープからグローバルスコープあるいは自分を囲むスコープに存在する変数への再代入を可能にする。

先ほどの「関数内でグローバル変数と同じ名前の変数に代入をしてみる」例では、グローバル変数の値が変わるのではなく、ローカルスコープに新たに名前が導入されたが、global キーワードを付けて変数を宣言することで（global 文）、その名前はグローバルスコープに存在するものとして取り扱われる。

global キーワードの使用例を以下に示す。

```
name = "insider.net"
def test():
  global name
  name = "windows server insider"
  print(locals())

test()
print(name)
```

global キーワードの使用例

global キーワードには続けて、グローバルスコープに存在するとして扱う名前を記述する。「global name = ……」のようにグローバル変数の宣言と代入（再束縛）を同時にはできないので注意しよう。

実行結果は次のようになる。

```
>>> name = "insider.net"
>>> def test():
...   global name
...   name = "windows server insider"
...   print(locals())
...
>>> test()
{}
>>> print(name)
windows server insider
```

ローカルスコープに名前 name はなく、グローバル変数 name の値が書き換えられた

関数内部での組み込み関数 locals の呼び出しを見ると分かる通り、ローカルスコープには name という名前は存在していない。その一方で「name = "windows server insider"」行によりグローバル変数 name の値が変化している（「windows server insider」に再束縛された）ことが分かる。

なお、上の例では最初にグローバル変数 name が確実に存在しているように「name = "insider.net"」行を実行しているが、global キーワードに続けて記述する名前は宣言時にグローバルスコープに存在している必要はない。よって、上の状態から以下のコードを実行してもエラーは発生しない（実行結果は割愛）。

```
del name
test()
print(name)
```

グローバル変数 name がなくても global キーワードの使用はエラーにはならない

→目次に戻る

もう 1 つのキーワード nonlocal はローカルスコープを囲むスコープに存在する名前に対して、ローカルスコープから代入を行えるようにするものだ。以下に例としてカウンターを作成する関数を示す。最初にエラーとなる例を示そう。

```
def makecounter():
  n = 0
  def count():
    n += 1
    return n
  return count

counter = makecounter()
print(counter())
```

カウンター関数を作成する関数 makecounter

これは典型的なクロージャであり、外側のスコープで定義されている変数 n の値を、内側の関数で参照、変更している。実際にこれを利用しようとすると次のようになる。

```
>>> def makecounter():
...   n = 0
...   def count():
...     n += 1
...     return n
...   return count
...
>>> counter = makecounter()
>>> print(counter())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in count
UnboundLocalError: local variable 'n' referenced before assignment
```

内側の関数 count 内では変数 n が未定義

内側の関数ではローカルスコープの外部にある名前を参照はできるが、その値を変更することはできない（上の global キーワードの例では、代入しようとすると同じ名前が新たにローカルスコープに導入されていた。今回は導入される前に、その値を変更しようとしている）。そのため、「n += 1」行は「未定義の変数の値を変更しようとした」と解釈されて、実行時に「UnboundLocalError」例外が発生している。こうした状況を避けるために nonlocal キーワードが利用できる。上のコードを修正すると次のようになる。

```
def makecounter():
  n = 0
  def count():
    nonlocal n
    n += 1
    return n
  return count

counter = makecounter()
print(counter())
print(counter())
print(counter())
```

修正後のコード

「nonlocal n」行で名前 n がローカルスコープを囲むスコープに存在していることを明示することで、その値を変更できるようになった。実際の実行結果を以下に示す（確かに動作しているかを確認するために「print(counter())」行の数も増やしている）。

```
>>> def makecounter():
...   n = 0
...   def count():
...     nonlocal n
...     n += 1
...     return n
...   return count
...
>>> counter = makecounter()
>>> print(counter())
1
>>> print(counter())
2
>>> print(counter())
3
```

内側の関数 count が変数 n の値を変更できていることが分かる

このようにクロージャから外側のスコープで定義されている変数の値を変更する必要があるといった場合には nonlocal は便利に使えるはずだ。また、ここでは組み込み関数 locals の呼び出しは行っていないが、「ローカルスコープを囲むスコープ」を作成したときの例と同様な結果となる（つまり、外側のスコープで定義されている変数が「自由変数」としてローカルな名前空間に存在するものとして扱われていることが分かる）。

◇ ◇ ◇ ◇ ◇ ◇

本章では Python におけるスコープと名前空間に関連する事項を取り上げた。次章では Python の例外を取り上げる。なお、VS の［Interactive］ウィンドウは本章で見たように変数 __builtins__ が参照するオブジェクトが通常の対話環境と異なるが、その一方で対話環境のスコープを __main__ モジュール以外のものに変更できる。興味のある方は「Working with the Python Interactive Window」（英語）などを参照されたい。

特集：Visual Studio で始める Python プログラミング

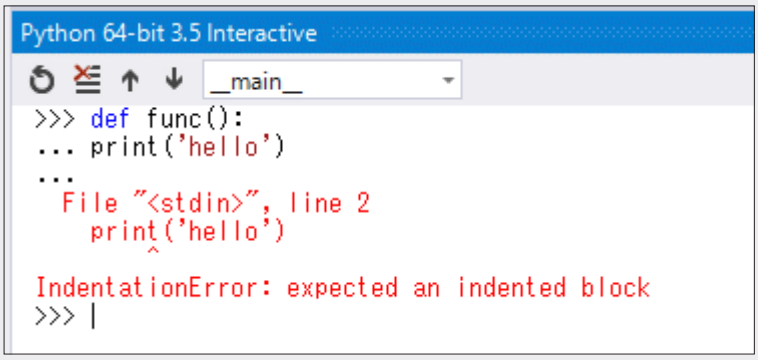
10.Python の例外をサクサク理解しよう

Python では構文エラーを含めて、大半のエラーを例外の形で取り扱う。そのための機構である try 文や raise 文などについてサンプル多めで見ていこう。

前章では、Python のモジュール／パッケージ／名前空間とスコープなどについて見てきた。本章では Python の例外処理の基本を見ていくことにしよう。

Python の例外

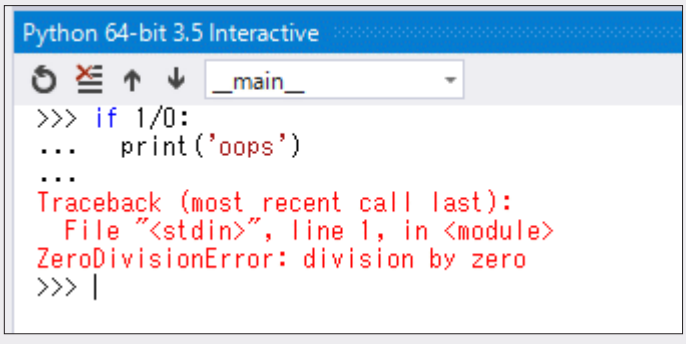
Python ではプログラム実行時に発生するエラー（および Python コードの構文解析時に発生するエラー）を例外の形で扱う。最も簡単な例外の例は次のようなものだろう（これは VS の［Interactive］ウィンドウで実行したものだ）。



インデントを間違えているので、IndentationError 例外が発生した

上の例では、関数定義内のブロックにインデントを付けていないために IndentationError という例外が発生している。ここで出てきた IndentationError クラスは構文エラーを表す SyntaxError クラスの派生クラスとなっている。

今見たのは Python コードの構文解析時に検出される例外だが、プログラム実行時に発生するエラーももちろん例外として取り扱われる。以下に例を示す。



ゼロ除算エラー

Python には多数の例外が組み込みで用意されている。全ての組み込み例外は BaseException クラスから派生する（ユーザー定義の例外クラスも作成でき、その場合は BaseException クラスから派生させずに、Exception クラスから派生させることが推奨されている）。実際の例外クラスのクラス階層は以下のようなコードを実行することで一覧表示可能だ（現在の実行コンテキストで定義されている例外を表示する。そのため、［Interactive］ウィンドウでの実行結果と「python ～」とコマンドラインから実行した場合の結果は異なる）。

```
def getsubcls(cls, n=0):
  if n == 0:  # 最初だけ特別にインデントなしでクラス名を表示
    print(cls.__name__)

  for subcls in cls.__subclasses__():  # 指定されたクラスの派生クラスを反復する
    print('  ' * (n+1), subcls.__name__, sep='')  # インデント付きでクラス名表示
    getsubcls(subcls, n + 1)  # そのクラスが派生クラスを持っていればそれらを表示

getsubcls(BaseException)
```

あるクラスの派生クラスを一覧表示する関数

コードの詳しい解説は省略するが、この関数 getsubcls ではパラメーターに渡された「クラス」オブジェクトに対して「__subclsses__」メソッドを呼び出し、その派生クラスを取得して、それに対して反復処理を行っている。反復処理内では、個々の派生クラス名の表示（インデント付き）とそのクラスの派生クラスを一覧表示している。［Interactive］ウィンドウでこれを実行すると、例えば次のような結果となる（抜粋。なお、コマンドプロンプトから python コマンドで対話環境を起動した場合には、空行の部分で関数定義が終了してしまうかもしれない。その場合は空行を含めないようにしよう）。

```
>>> def getsubcls(cls, n=0):
...   if n == 0:
...     print(cls.__name__)
...
...   for subcls in cls.__subclasses__():
...     print('  ' * (n+1), subcls.__name__, sep='')
...     getsubcls(subcls, n + 1)
...
>>> getsubcls(BaseException)
BaseException
  Exception
    SystemError
      CodecRegistryError
    MemoryError
    Error
      … 省略 …
    ArithmeticError
      FloatingPointError
      OverflowError
      ZeroDivisionError
    LookupError
      … 省略 …
    EOFError
    NameError
      … 省略 …
    SyntaxError
      IndentationError
        TabError
      … 省略 …
    RuntimeError
      RecursionError
      NotImplementedError
        … 省略 …
  GeneratorExit
  SystemExit
  KeyboardInterrupt
```

上のコードの実行結果

上で見た IndentationError クラスや ZeroDivisionError クラスもこの階層に含まれていることが分かる。また、クラスの継承関係をインデントで表現していることにも注意されたい。例えば、IndentationError クラスは構文エラー一般を表す SyntaxError クラスの派生クラスだ（同様に ZeroDivisionError クラスは ArithmeticError クラスの派生クラス）。

例外が発生した場合には、それを処理しない限りはプログラムの実行は中断して、例外の発生箇所を示すスタックトレースと発生した例外の種類が表示される。例外を処理するための構文が try 文である。次ページではこれについて見ていこう。

try 文

Python での例外処理には try 文を使用する。以下では、まずはその基本型を見た後に、except 節での細かな指定の方法などを見ていく。

try 文の基本型

基本となる構文を以下に示す。最もシンプルなのは try 節と except 節でだけ構成されるが、ここでは finally 節も含めて紹介しよう。

```
try:
  # 例外を発生する可能性があるコード
except:
  # 例外発生時に実行するコード
finally:
  # 例外の有無に関わらず最後に実行されるコード
```

例外処理の基本構文

try 節には例外を発生する可能性があるコードを記述する。そこで例外が発生した場合には except 節がこれをキャッチするので、そこで例外を処理する。finally 節は例外の有無に関係なく、try 文の終了時に実行される。このことから、finally 節では必ず行わなければならない後処理を書くことが多い。

ところで、先ほどの関数 getsubcls はパラメーターにクラスオブジェクトを受け取ることを前提としている。そこで未定義のクラス「SomeException」をこの関数に渡してみよう。

```
>>> getsubcls(SomeException)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'SomeException' is not defined
```

存在していないクラス名を入力すると NameError 例外が発生する

そして、上で見た基本構文を使って、これを処理するコードは次のようになる。関数 getsubcls には「BaseException」ではなく「SomeException」を渡している点に注意。

```
try:
  getsubcls(SomeException)
except:
  print('specify defined class name')
finally:
  print('finally clause')
```

エラーを発生させ、それを処理

これを［Interactive］ウィンドウで実行した結果を以下に示す。

```
>>> try:
...   getsubcls(SomeException)
... except:
...   print('specify defined class name')
... finally:
...   print('finally clause')
...
specify defined class name
finally clause
```

例外が処理された

例外が発生したため、except 節でこれがキャッチされてメッセージが表示され、その後に finally 節で「finally clause」というメッセージが表示されている。余談だが、この例外は関数 getsubcls 内部で発生したものではなく、関数呼び出しの時点で「SomeException」という名前がなかったことから発生している。

そして、「SomeException」ではなく、既存の例外クラスである「ArithmeticError」を渡した場合の実行結果は次のようになる。

```
>>> try:
...   getsubcls(ArithmeticError)
... except:
...   print('specify defined class name')
... finally:
...   print('finally clause')
...
ArithmeticError
  FloatingPointError
  OverflowError
  ZeroDivisionError
finally clause
```

例外が発生しなくても finally 節は実行される

ArithmeticError クラスは実際に存在するクラス（最初に見た ZeroDivisionError クラスの基底クラス）であり、この場合は例外は発生せずにその派生クラスが表示されている。そして、実行結果を見れば分かる通り、例外が発生しなくとも finally 節が実行される。

キャッチする例外の指定

上のコードでは except 節に何も指定していないが、これは「ワイルドカードの except 節」と呼ばれるもので、例外なら何でもキャッチしてしまう。これは Python ではあまり推奨されておらず、except 節でキャッチする例外を個別に指定していき、「ワイルドカードの except 節」を記述するとしたら最後にする。以下に except 節に

キャッチする例外を指定した例を示す（「ワイルドカードの except 節」がないとどうなるか見るため、ここではあえてこれを指定していない）。

```
def func(exp):
  try:
    print(eval(exp))
  except ZeroDivisionError:
    print('division by zero')
  except NameError:
    print('undefined name')
```

キャッチする例外を指定

関数 func に文字列を渡すと、それが評価される。実際に実行した例を以下に示す。ここでは「1/0」のゼロ除算、「undefinedvar」という未定義の名前に 2 を加算、数値の「1」に文字列の「100」を加算の 3 つの計算を行っている。

```
>>> def func(exp):
...   try:
...     print(eval(exp))
...   except ZeroDivisionError:
...     print('division by zero')
...   except NameError:
...     print('undefined name')
...
>>> func('1/0')
division by zero
>>> func('undefinedvar + 2')
undefined name
>>> func('1 + str(100)')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in func
  File "<string>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

「ワイルドカードの except 節」がないので、異なる型の値の加算はキャッチされなかった

最初の 2 つに関しては except 節で例外を指定しているのでキャッチされるが、最後のものについては指定がないので、処理されずに Python がスタックトレースと発生した例外を表示している。「ワイルドカードの except 節」があればこうした例外もキャッチできる（が、意図せずに発生した例外の原因がプログラムミスにあるとすれば、何でもかんでも「ワイルドカードの except 節」でキャッチしてしまうのがよいかには議論の余地があるだろう）。

このとき注意したいのは、例外クラスを指定する順番だ。例えば、上のコードを次のように修正したとする。

```
def func(exp):
  try:
    print(eval(exp))
  except Exception:
    print('some exception!')
  except ZeroDivisionError:
    print('division by zero')
  except NameError:
    print('undefined name')
```

キャッチする例外を指定する際には、その順番に注意する必要がある

ここでは Exception クラスを真っ先に指定しているが、こうすると ZeroDivisionError 例外が発生しようが、NameError 例外が発生しようが、最初の except 節でキャッチされてしまう（実行例は割愛）。これは 2 つの例外クラスが Exception クラスの派生クラスとなっているからだ（これら 2 つの例外クラスが Exception クラスと「互換性がある」あるいは「is-a の関係」にあるため）。

また、例外を指定した場合には、例外クラスに続けて「as 変数名」とすることで、その例外を表すオブジェクトを変数に代入できる。以下に例を示す。ここでは上の関数 func 呼び出しで指定した引数を要素としたリストに対して反復処理を行って、その評価を行っている。

```
for item in ['1/0', 'hoge + 2', '1 + str(100)']:
  try:
    print(eval(item))
  except ZeroDivisionError as e:
    print('devision by zero:', e)
  except NameError as e:
    print('undefined name:', e.args)
  except TypeError as e:
    print('incompatible types:', e, e.args)
```

例外オブジェクトを取得する

このコードでは「as e」として変数 e に例外オブジェクトを取得している。例外オブジェクトのメンバ「args」には通常、エラーメッセージが格納されている。そのため「e.args」とすればエラーメッセージにアクセスできるが、単に「e」とするだけでもこれを取得できるように準備されている。その違いが分かるように、3 つの except 節にある関数 print 呼び出しではそれぞれ「e」のみ、「e.args」のみ、その両方を出力するようにしてある。実行結果は次のようになる。

```
>>> for item in ['1/0', 'hoge + 2', '1 + str(100)']:
...   try:
...     print(eval(item))
...   except ZeroDivisionError as e:
...     print('devision by zero:', e)
...   except NameError as e:
...     print('undefined name:', e.args)
...   except TypeError as e:
...     print('incompatible types:', e, e.args)
...
devision by zero: division by zero
undefined name: ("name 'hoge' is not defined",)
incompatible types: unsupported operand type(s) for +: 'int' and 'str'
  ("unsupported operand type(s) for +: 'int' and 'str'",)
```

実行結果

出力結果が 4 行あるように見えるが、これは 3 つ目の出力結果に改行を挿入したため。

「e.args」ではエラーメッセージがかっこに囲まれて出力されているが、その内容は「e」と同じことが分かる。

else 節

else 節についても簡単に触れておこう。Python では while 文などでも else 節を記述できたが、try 文でも else 節を書ける。else 節は try 節で例外が発生しなかった場合に実行される（else 節とは異なり、finally 節は例外の有無に関係なく実行される）。

先ほどの関数 func を次のように変更して、その動作を確認してみよう。

```
def func(exp):
  try:
    print(eval(exp))
  except ZeroDivisionError:
    print('division by zero')
  except NameError:
    print('undefined name')
  except Exception:
    print('some exception!')
  else:
    print('success!')
  finally:
    print('finally clause')
```

関数 func に else 節と finally 節を追加

実行すると次のようになる。エラーなしで try 文が実行されたときには、以下で強調書体としている「success!」から分かるように else 節が実行される。

```
>>> def func(exp):
...   try:
...     print(eval(exp))
...   except ZeroDivisionError:
...     print('division by zero')
...   except NameError:
...     print('undefined name')
...   except Exception:
...     print('some exception!')
...   else:
...     print('success!')
...   finally:
...     print('finally clause')
...
>>> func('1 + 1')  # 成功する場合
2
success!          # else節が実行された
finally clause    # finally節が実行される
>>> func('1 / 0')  # 失敗する場合
division by zero
finally clause    # else節は実行されずにfinally節が実行される
```

例外が発生しなければ else 節も実行される

最後に例外の送出についても簡単にまとめておこう。

例外の送出

ここまでプログラム実行時に発生した例外の処理を見てきたが、raise 文を使用することで、任意の例外を任意の時点で送出できる。

最も分かりやすい例を以下に示す（これまでと同様、［Interactive］ウィンドウでの実行結果）。

```
>>> raise RuntimeError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError
```

RuntimeError 例外の送出

このとき、コンストラクタにエラーメッセージを引き渡すこともできる（強調書体部分に注目）。

```
>>> raise RuntimeError('hello from runtime error')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: hello from runtime error
```

エラーメッセージの指定

例外処理時に、その例外を適切に処理できなかったときには単に「raise」とするだけで、例外が再送出される。以下に例を示す。ここでは関数 func が例外を処理せずに、再送出するようにしている。また、関数 caller から関数 func を呼び出すことを想定している。関数 caller では例外処理をしていないことに注意。

```
def func(exp):
 try:
  return eval(exp)
 except ZeroDivisionError:
  print('division by zero')
  raise
 except NameError as e:
  print('undefined name')
  raise
 except Exception as e:
  print('some exception!', e)
  raise RuntimeError(e.args) from e

def caller():
 x = input('input expression:')
 print(func(x))
```

修正した関数 func とそれを呼び出す関数 caller

最後の「except Exception from e」行では、発生した例外からさらに例外を発生させている（例外の連鎖）。このときには、新たに発生した例外の原因を「from」に続けて記述する。

実行例を以下に示す。なお、ここでは上のコードを exceptionsample.py ファイルに保存し、これを［Interactive］ウィンドウでインポートしている（［Interactive］ウィンドウでうまく動かないときには Visual Studio を再起動するか、コマンドラインの python コマンドで対話環境を起動するなどしてみてほしい）。

```
>>> from exceptionsample import func, caller
>>> caller()  # 例外の再送出
input expression:
1/0

division by zero
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".¥exceptionsample.py", line 16, in caller
    print(func(x))
  File ".¥exceptionsample.py", line 3, in func
    return eval(exp)
  File "<string>", line 1, in <module>
ZeroDivisionError: division by zero
>>> caller()  # 例外の連鎖
input expression:
1 + str(100)

some exception! unsupported operand type(s) for +: 'int' and 'str'
Traceback (most recent call last):
  File ".¥exceptionsample.py", line 3, in func
    return eval(exp)
  File "<string>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".¥exceptionsample.py", line 16, in caller
    print(func(x))
  File ".¥exceptionsample.py", line 12, in func
    raise RuntimeError(e.args) from e
RuntimeError: ("unsupported operand type(s) for +: 'int' and 'str'",)
```

実行例

最初の例（「1/0」）では、exceptionsample.py ファイルの 3 行目（「eval(exp)」行）で発生した例外がそのまま再送出されている。一方、2 つ目の例では、同様に 3 行目で発生した TypeError 例外を直接の原因として RuntimeError 例外が送出されたことが分かる（スタックトレースが 2 つあり、その後者の「raise RuntimeError(e.args) from e」で RuntimeError 例外が送出されたことが分かる）。関数 caller 側では元の関数 func と同様にして再送出された例外を処理できる（コードは割愛）。

独自の例外クラスも作成できる。以下に簡単な例を示す。

```
class MyException(Exception):
  def __init__(self, message):
    self.message = message

try:
  raise MyException('hello')
except MyException as e:
  print(e)
```

独自の例外クラスの作成

ポイントは Exception クラスまたはその派生クラスを、独自クラスの基底クラスとすることだ。Python ではユーザー定義の例外クラスは Exception クラスの派生クラスとする必要があるので注意しよう（実行結果は割愛）。

◇ ◇ ◇ ◇ ◇ ◇

本章では、Python の例外の基礎について見てきた。Python では例外が try ～ except ～ else ～ finally 文で取り扱う。except ではキャッチする例外を指定したり、例外オブジェクトを取得したりできる。else 節は try 節で例外が発生されなかったときに実行され、finally 節は try 節での例外の有無に関係なく実行される。各節がどのような順序で実行されるかは（本章では触れなかったが、例外の再送出が絡んだ場合にはより一層の）注意が必要だ。次章では、落ち穂拾い的にさまざまな要素を取り上げていくことにしよう。

特集：Visual Studio で始める Python プログラミング

11.Python の文字列／ファイル操作／組み込み関数（もしくは落ち穂拾い）

Python でプログラミングをする上で必須の知識といえる文字列やファイルの扱い方の基本、便利に使える組み込み関数を本項では紹介する。

本章でこれまでに取り上げてこなかったが、覚えておくべき事項として Python の文字列の基本、ファイル操作の基本、幾つかの組み込み関数の使い方を見ていこう。

文字列

本書では整数、浮動小数点数など、Python の基本データ型については明確に取り上げてこなかった（読者の多くはこうしたデータ型の概念については既にご存じであろうという考えからだ）。文字列もそうしたデータ型の 1 つではあるが、Python 独自の文字列の使い方もあるので、以下ではこれについて見ていくことにしよう。なお、Python の組み込み型については「TensorFlow や Chainer に興味があるけど、Python 未経験の技術者が最低限知っておいた方がいい基礎文法まとめ (2/6)」にまとめてあるので、そちらを参照してほしい。

まず簡単に文字列についてまとめておこう。

- 文字列はシングルクオート／ダブルクオート、あるいはこれらを 3 つ連続したもので囲む
- Python では文字列は Unicode 文字が連続したもの（シーケンス）として扱われる
- 関数／クラス／モジュールでは「docstring」と呼ばれる特別な使い方がされる

文字列の基本

以下に簡単な例を示す。

```
>>> s = 'It¥'s Python that is "most suitable script language" for Visual Studio'
>>> s = "It's Python that is ¥"most suitable script language¥" for Visual Studio"
```

ダブルクオートとシングルクオートで囲んだ文字列

見ての通り、シングルクオート内ではダブルクオートをエスケープせずに使用できるが、シングルクオートを使うには「¥」でエスケープする必要がある。逆も同様だ。また、「¥t」でタブ文字などの一般的なエスケープシーケンスも利用可能だ。また、8 進数値による文字指定（「¥xxx」。「xxx」は 8 進数値）、16 進数値による文字指定（「¥x」）、Unicode のコードポイントによる指定（「¥u」「¥U」）、Unicode 名による指定（「¥N{Unicode

名}|）なども可能だ。これらについては Python のドキュメント「2.4.1. 文字列およびバイト列リテラル」などを参照してほしい。

なお、エスケープ文字（¥）をエスケープなしで文字列内に記述するには文字列を囲む最初のシングルクオート／ダブルクオートの前に「r」を付加する。

```
path = r"c:¥project¥python"
path
print(path)
```

「raw string」と呼ばれる文字列の記述方法

このような文字列を「raw string」と呼ぶ。上のコードを対話環境で実行すると、次のようになる。

```
>>> path = r"c:¥project¥python"
>>> path
'c:¥¥project¥¥python'
>>> print(path)
c:¥project¥python
```

r"" 形式の文字列を対話環境で評価、組み込み関数 print で表示した結果

raw string では、「¥n」のような文字が含まれていても、それはエスケープされた文字ではなく「¥」と「n」という文字が連続しているだけであることに注意しよう。以下に対話環境での実行例を示す。

```
>>> s = 'f¥tbar'
>>> print(s)  # 「¥t」はタブ文字を表す
f	bar
>>> s = r'f¥tbar'
>>> print(s)  # 「¥t」は「¥」と「t」という文字の並び
f¥tbar
```

raw string を扱う上での注意点

文字列はシーケンスなので、次のように for ループで 1 文字ずつその要素を取り出せる。

```
s = "Python"
for c in s:
  print(c)
```

文字列に対して反復処理を行う

実行結果がどうなるかはお分かりだろうが、一応示しておこう。

```
>>> s = "Python"
>>> for c in s:
...   print(c)
...
P
y
t
h
o
n
```

実行結果

トリプルクオート（三重引用符）は改行を含む文字列を記述する際に利用する。以下に例を示す。トリプルクオートでは文字列は「'''」または「"""」で囲まれるので、途中に単独の「'」や「"」を自由に含めることができる。また、複数行にわたる文字列を自由に記述できることから、次に見る docstring などで使われることが多い。

```
s = """It's Python that is
"most suitable script language" for Visual Studio"""
print(s)
```

トリプルクオート

文字列と文字列を空白文字で区切って記述した場合、それらは 1 つの文字列となる。

```
s = "insider" ".net"
print(s)  # 出力結果： insider.net
```

空白文字で区切られた文字列リテラルは単一の文字列リテラルになる

なお、文字列を操作するメソッドや数値やリストを文字列化する組み込み関数などについては「TensorFlow や Chainer に興味があるけど、Python 未経験の技術者が最低限知っておいた方がいい基礎文法まとめ (2/6)」を参照のこと。

docstring

Python では docstring（ドキュメンテーション文字列）と呼ばれる文字列の使い方をすることで、関数／クラス／モジュールにドキュメントを付加することがとても簡単に行えるようになっている。docstring とは簡単に言ってしまえば「関数／クラス／モジュール定義の先頭の式がリテラル文字列の場合、それがそのドキュメントとして扱われる」というものだ。以下に例を示す。

```
def hello():
  '''Print "Hello world"'''
  print('Hello world')

hello()
help(hello)
hello.__doc__
```

docstring 付きの関数 hello

関数 hello の先頭の式がリテラル文字列となっているので、これは docstring として扱われる。docstring は関数 hello の属性「__doc__」として保存され、外部から参照が可能だ。例えば、組み込み関数 help に関数 hello を渡すと、これが参照され、ヘルプドキュメントとして表示される。以下に上のコードの実行例を示す。

```
>>> def hello():
...   '''Print "Hello world"'''
...   print('Hello world')
...
>>> hello()
Hello world
>>> help(hello)
Help on function hello in module __main__:

hello()
    Print "Hello world"  # docstringがヘルプドキュメントとして表示されている

>>> hello.__doc__  # docstringはそれを記述した関数などの属性「__doc__」に保存される
'Print "Hello world"'
```

docstring の利用例

関数やクラス、モジュールの先頭にはコメントなどの形で、それらが何のためのものかなどを記述することがよくあるが、Python ではこれを docstring という形式で言語機構内に含めている。そして、docstring を使うことで、これが自動的にヘルプドキュメントとして使えたり、プログラムの他の部分から参照できたりするようになるので、活用するようにしよう。なお、docstring をどう書くべきかについては Python のドキュメント「4.7.6. ドキュメンテーション文字列」を参照してほしい。

ファイル操作

Python におけるファイル操作の基本型は、C 言語におけるそれとよく似ている。つまり、関数 open でファイルを開き（ファイルオブジェクトが返される）、ファイルに対して何らかの処理を行い、ファイルオブジェクトに対する close メソッド呼び出しでクローズする。ファイルを開く際にはファイル名とともにモード（読み込み／書き込み／テキストモード／バイナリモードなど）を指定する。

ファイル操作の基本

以下に簡単な例を示す。ここでは、ファイルをまず作成し、そこにテキストを書き込んでクローズしてから、その内容を読み込んでみる。

```
f = open('sample.txt', 'w')  # 書き込みモードでテキストファイルをオープン
f.write('insider.net')       # テキストの内容をwriteメソッドで書き込み
f.close()                    # ファイルをクローズするにはcloseメソッドを使用
f = open('sample.txt', 'r')  # 読み込みモードでオープン
t = f.read()                 # ファイルの内容を読み込み
print(t)
```

ファイル操作の基本

対話環境での実行結果は次のようになる。

```
>>> f = open('sample.txt', 'w')
>>> f.write('insider.net')  # writeメソッドは書き込んだ文字数を返送する
11
>>> f.close()
>>> f = open('sample.txt', 'r')
>>> t = f.read()
>>> print(t)
insider.net
```

対話環境での実行結果

組み込み関数 open の第 1 引数にはこれから操作を行うファイルの名前を、第 2 引数にはオープンするモードを指定する。最初の open 呼び出しではファイル名に「sample.txt」を、モードには「'w'」を指定している。「'w'」は書き込みモードを意味する。指定可能なモードとして以下のものがある。

モード	意味
w	書き込み
r	読み込み
x	書き込み。指定したファイルが既に存在している場合は例外が発生
a	追記
b	バイナリモード
t	テキストモード
+	更新用に開く（例：「'r+'」「'a+'」で読み書き両用にオープン可能）

組み込み関数 open で指定可能なモード

デフォルトではテキストモードでオープンされるので、これは省略可能だ（上では「'w'」「'r'」としているが、これは「'wt'」「'rt'」の省略形であり、テキストモードでファイルを開いている）。またテキストモードでは、指定されたエンコーディング方式（デフォルトではプラットフォーム依存）を用いてファイルが読み書きされるとともに、行末コードの変換などが行われる。対してバイナリモードでは、そのような変換が行われずに、データはバイト列として扱われる。

この例では、ファイルへの書き込みには write メソッドを使用している。このメソッドは「書き込んだ文字数」を返送する。そして、ファイル操作が終わったら、忘れずに close メソッドを呼び出す必要がある（が、これを省略して、ファイルのオープン／クローズが対になるような構文も用意されている。後述）。

ファイルの読み込みには read メソッドを使用している。read メソッドはファイルの内容を全て読み込むものだが、1 行ごとにファイルを読み込む場合には readline メソッドを利用できる。以下に簡単な例を示す。

```
f = open('sample.txt', 'w')  # 複数行からなるファイルの作成
f.write('insider.net¥nbuildinsider')
f.close()
f = open('sample.txt', 'r')  # 読み込みモードでオープン
f.read()                     # 全内容を読み込み
f.close()
f = open('sample.txt')       # 第2引数を省略すると読み込みモードでオープン
f.readline()                 # 1行ずつ読み込み
f.readline()                 # 1行ずつ読み込み
f.close()
```

複数行の読み込み

対話環境での実行結果を以下に示す。

```
>>> f = open('sample.txt', 'w')
>>> f.write('insider.net¥nbuildinsider')
24
>>> f.close()
>>> f = open('sample.txt', 'r')
>>> f.read()
'insider.net¥nbuildinsider'
>>> f.close()
>>> f = open('sample.txt')
>>> f.readline()
'insider.net¥n'
>>> f.readline()
'buildinsider'
>>> f.close()
```

対話環境での実行結果

ループを使って、ファイルの内容を逐次読み込んでいくことも可能だ。

```
f = open('sample.txt')
for line in f:
  print(line)
f.close()
```

ファイルオブジェクトをシーケンスとして利用

このようにすると、ループごとにファイルの内容が 1 行ごとに読み込まれる（実行結果は割愛）。

組み込み関数 print でのファイルへの書き込み

本章ではコンソールへの出力に組み込み関数 print を使用してきた。この関数はデフォルトでは標準出力に出力を行うというだけで、実際には上で見たようなファイルへ出力を行うことも可能だ。以下に例を示す。

```
f = open('sample.txt', 'w')
print('hello world', file=f)  # sample.txtファイルへの出力
f.close()
f = open('sample.txt')
f.read()
f.close()
```

標準出力からファイルへの切り替え

with ステートメント

with ステートメントを使うと、組み込み関数 open で開いたファイルを close メソッドで閉じるという定型処理を抽象化できる。典型的な例を以下に示す。

```
f = open('sample.txt', 'w')    # 複数行からなるファイルの作成
f.write('insider.net¥nbuild insider')
f.close()
with open('sample.txt') as f:  # オープンしたファイルを表すオブジェクトをfで受け取る
  for line in f:               # 後は上と同様。だが、f.close呼び出しは必要ない
    print(line)
```

with ステートメントの利用例

with ステートメントを使うと、ファイル操作の終了時には必ずファイルがクローズされるようになる。close メソッドを呼び出すのを忘れることがなくなるのは 1 つのメリットだ。しかし、これにはもう 1 つ大きなメリットがある。with ステートメントを使えば、ファイル操作時に例外が発生しても必ずクローズできるのだ。ファイル操作に例外がつきものであることを考えると、実際のファイル操作は（with ステートメントを使わないと）次のようになる。

```
try:
  f = open(……)
  # ファイル操作
finally:
  f.close()
```

ファイル操作を try ～ finally 文で囲み、確実にファイルがクローズされるようにする

上で見たように with ステートメントを使えば、try ～ finaly 文で囲まずとも確実にファイルをクローズできるようになる。

ここではファイル操作の基本だけを見てきたが、Python にはシリアライズ／デシリアライズを行うための pickle モジュールや、JSON 形式のデータを扱うための json モジュールなど、入出力を簡潔に記述するためのさまざまなモジュールも用意されているので、興味のある方は、Python のドキュメントを参照してほしい。

本章の最後に Python の組み込み関数で便利に使えるものを幾つか紹介していこう。

組み込み関数を幾つか

以下では Python の組み込み関数で便利に使えるものをカテゴリに分けて幾つか見ていく。

反復可能なオブジェクトに対する処理

反復可能なオブジェクトの要素に対して、何らかの処理を行う関数としては次のものがある。

関数	説明
all	全ての要素が真と評価されたらTrueを、そうでなければFalseを返す（反復可能オブジェクトが空の場合はTrue）
any	全ての要素のいずれかが真と評価されたらTrueを、全てが偽と評価されたらFalseを返す（反復可能オブジェクトが空の場合はFalse）
filter	第1引数に渡した関数で、各要素を評価し、その結果が真となる要素からなるイテレータを返送する
len	要素数を返す
map	第1引数に渡した関数を各要素に適用するようなイテレータを返送する
reversed	要素を逆順に返送するイテレータを返送する
sorted	各要素をソートした結果を格納する新たなリストを返送する（元の反復可能オブジェクトは変更されない）。ソートに使用する関数も指定可能
sum	各要素の総和を計算する
zip	引数に渡した複数の反復可能オブジェクトの各要素を組み合わせたイテレータを返送する

反復可能なオブジェクトに対する処理

例を以下に示す。各行のコメントには対話環境での実行結果（評価結果）と必要ならかっこ内に説明を付記している。

```
all([0, 1, 2])          # False
all([1, 2, 3])          # True
all([])                 # True
any([0, 1, 2])          # True
any([None, True, False])  # True
any([])                 # False
list(filter(lambda x: x % 2 == 0, list(range(5))))  # [0, 2, 4]（偶数を抽出）
len(list(range(10)))    # 10（10要素からなるリストの要素数）
list(map(lambda x: x * x, list(range(5))))  # [0, 1, 4, 9, 16]（各要素を二乗したリスト）
list(reversed(range(5)))  # [4, 3, 2, 1, 0]
sum([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])  # 55
list(zip([1, 2, 3], ['a', 'b', 'c', 'd']))  # [(1, 'a'), (2, 'b'), (3, 'c')]
```

反復可能なオブジェクトに対する処理

文字列関連

文字列に関連する組み込み関数としては次のものがある。

関数	説明
chr	指定した引数をUnicodeコードポイントとするUnicode文字を返す
ord	指定した文字のUnicodeコードポイントを返す
eval	引数に渡した文字列をPython式として解析／評価する
exec	引数に渡した文字列を一連のPython文として解析／実行する

文字列に関連する組み込み関数

組み込み関数 chr ／ ord は対となる機能を提供する。組み込み関数 eval ／ exec の違いは前者が Python の式（単一の式、あるいは「x if y if z」のような条件式の連続）を解析／評価するのに対して、後者は Python の文（例えば、関数定義など）を解析／実行する点だ（複雑な処理を動的に実行するのであれば組み込み関数 exec を使用する）。これら 2 つの関数は文字列だけではなく、組み込み関数 compile で作成可能なコードオブジェクトも引数に受け取る（が、本稿では取り上げない。興味のある方は「compile」などを参照のこと）。

以下に例を示す。先ほどと同様、対話環境での実行結果（評価結果）をコメントに記す（説明はかっこ内に記した）。

```
ord('ほ')     # 12411
chr(12411)    # 'ほ'
hex('ほ')     # '0x307b'
chr(0x307b)   # 'ほ'
import random  #  （乱数を利用）
flag = random.randint(0, 9) % 2  #  （1/2で真か偽）
eval('"insider.net" if flag else "build insider"')  #  （変数flagの値次第）
eval('def foo():  print("hello")')  #  （組み込み関数evalでは評価できない）
exec('def foo():  print("hello")')  #  （組み込み関数execはこれを実行できる）
```

文字列に関連する組み込み関数

◇ ◇ ◇ ◇ ◇ ◇

本章では（VS ユーザーに向けて）Python プログラミングを始めるに当たって必要となりそうな事項を取り上げてきた（とはいえ、本書で取り上げてきた要素は VS を使っていない Python 初心者の方々にも有益だと考えている）。他にも説明すべき事項は多々あるが、これまでに取り上げてきた要素をマスターすれば、取りあえず他者のコードを読み進めたり、自分でちょっとした Python コードを書いたりしながら、Python への理解を深めることができるようになるはずだ。

特集：Python 3.6 の新機能

Python 3.6 で追加された新機能をザックリ理解しよう

2016 年 12 月に Python の最新版であるバージョン 3.6 がリリースされた。ここではその中でも特徴的な新機能を幾つか紹介していく。

2016 年 12 月 23 日に Python の最新版であるバージョン 3.6 がリリースされた。本項ではその中でも特徴的な新機能を幾つか紹介していく。

Python 3.6 で追加された主要な新機能

Python のドキュメント「What's New In Python 3.6」や、そのリリースノート（英語）には Python 3.6 で追加された機能や、変更点がまとめられている。

大まかなところを拾っていくと、構文的には以下のような機能が追加されている。

- フォーマット文字列リテラル：いわゆる文字列補間
- 数値リテラルでのアンダースコアの使用：「123456」を「123_567」などと記述可能
- 型注釈：関数のパラメーター／戻り値の型に加えて、変数にも型注釈を付加可能に
- 非同期ジェネレータ：Python 3.5 の非同期イテレータに加えて、非同期ジェネレータもサポートされた
- 非同期内包表記：「async for」「await」などを内包表記中に記述可能に

本項ではサンプルコードとともに、上記について見ていくことにしよう。これ以外の変更点や改善点については前述のリンクなどを参照されたい。

フォーマット文字列リテラル

フォーマット文字列リテラルとは、他の言語においても最近になって実装されることが多くなったいわゆる「文字列補間」の Python バージョンのことだ（PEP 498。「PEP」は「Python Enhancement Proposal」の略で「Python の機能追加や改善策に関する提案」のこと）。Python では「f-string」「f 文字列」などと呼ばれることもある。

「f 文字列」という名前から想像できるかもしれないが、フォーマット文字列リテラルは文字列を囲むクオーテーションの前に「f」を前置し、その内部では「{ 式 }」と記述することで、その部分にその式の値（例えば、変数の値）を埋め込める。

以下に例を示す。

```
name = 'insider.net'
print(f'name: {name}')      # 出力結果： name: insider.net
print(f'2 ** 4 = {2 ** 4}') # 出力結果： 2 ** 4 = 16
```

フォーマット文字列リテラル

「出力結果」をコメントとして含めてあるが、これを見ると「{}」内に記述した変数の値や計算結果が補間されているのが分かる。特に説明はいらないだろう。なお、「{}」内には以前より Python で使われてきた書式指定文字列も記述可能だ（「:」以降に記述）。以下に例を示す。

```
print(f'2 ** 12 = {2 ** 12:>+8,d}') # 出力結果： 2 ** 12 =  +4,096
print(f'2 ** 12 = {2 ** 12:#_b}')   # 出力結果： 2 ** 12 = 0b1_0000_0000_0000
```

書式指定文字列の利用例

最初の例は「2 ** 12」の評価結果を「右寄せ（>）、正負両方の値で符号を付加（+）、8 桁幅（8）、3 桁ごとにカンマ（,）を挿入（,）、10 進数（d）」で表記することを指定するものだ。次の例では同じく「2 ** 12」の評価結果を「0b を前置（#）、4 桁ごとにアンダースコア（_）を挿入、2 進数（b）」で表記するように指定している（「#」指定については、補間する値の型によって意味が異なってくるので注意）。なお、アンダースコアを挿入できるのは、Python 3.6 の新機能の 1 つでもある。

書式指定文字列の構文の詳細については Python のドキュメント「書式指定文字列の文法」「書式指定ミニ言語仕様」などを参照してほしい。

数値リテラルでのアンダースコアの利用

数値リテラル内にアンダースコア（_）を記述できるようになった（PEP 515）。これは、桁数の多い数値リテラルの可読性を高めるためのものだ。アンダースコアは基数の指定子（「0b」「0o」「0x」）の直後、数字と数字の間に挿入できる。ただし、リテラルの先頭と末尾に置くことはできず、連続したアンダースコアは認められない。以下に例を示す。

```
num = 0x_FFFF_FFFF
print(f'0xffffffff = {num:,}')      # 出力結果： 0xffffffff = 4,294,967,295
num = 4_294_967_296
print(f'4_294_967_296 = {num:#_x}') # 出力結果： 4_294_967_296 =
0x1_0000_0000
#num = 1000_   # エラー（末尾のアンダースコア）
#num = _1000   # エラー（先頭のアンダースコア）
#num = 1__000  # エラー（連続したアンダースコア）
```

数値リテラルでのアンダースコアの利用

なお、前節でも見た書式指定文字列内でのアンダースコアの利用は、数値リテラル内でのアンダースコアの利用を提案した PEP 515 で規定されている。これを指定した場合、10 進表記の場合は 3 桁ごとに、2 進／8 進／16 進表記の場合は 4 桁ごとにアンダースコアが挿入されるようになっている。

型注釈／型ヒント機能の拡張

型注釈あるいは型ヒントと呼ばれる機能は Python 3.5 で、関数の引数と戻り値型について導入されたものだが（PEP 484）、Python 3.6 ではこれが拡張されて、変数（クラス変数、インスタンス変数を含む）についても型注釈を付加できるようになった（PEP 526）。

型注釈の詳細については割愛するが、基本的なことをいえば変数やパラメーターの名前に続けて「: 型名」と記述する。関数の戻り値については「-> 型名」で指定する。以下に例を示す。

```
def foo(num: int) -> str:
  return str(num)

print(foo('string'))  # エラー
print(foo(100))       # OK

num = '100'
print(foo(num))       # エラー
```

型注釈／型ヒントの利用例（thint.py ファイル）

型注釈／型ヒントはあくまでも静的な型解析を行うためのものであり、python コマンドで上記のスクリプトを実行する際には無視される。型解析を行うには mypy などのツールを使用する必要がある。上のコードには型に関するエラーが含まれてはいるが、実行は可能だ。以下に実行結果を示す。なお、このコードでは関数のパラメーターと戻り値型にのみ型注釈を付加しているので、Python 3.5 ／ 3.6 のいずれでも実行可能だ。

```
> python thint.py
string
100
100
```

型注釈を含む Python コードの実行

先ほども述べたように、型チェックを行うには mypy などのツールが必要だ。mypy をインストールするには pip コマンドを使用する。

```
> pip install -U mypy-lang
> pip install -U "typed_ast >= 0.6.3, < 0.7.0"
```

mypy と Python 3.6 構文を解釈させるために必要な typed_ast パッケージのインストール

本項の執筆時点（2017 年 2 月 23 日）では、typed_ast パッケージの最新版をインストールすると、Python 3.6 の構文をうまく解析できなかったので上記のコマンドラインではバージョンを指定してインストールしているが、これは将来的に解消されるはずだ（あるいは「pip install -U mypy-lang」コマンドでうまく依存関係を調整できるかもしれない）。

mypy をインストールしたら、コマンドラインで静的型チェックを行えるようになる。以下に例を示す。

```
> mypy thint.py
thint.py:4: error: Argument 1 to "foo" has incompatible type "str"; expected
"int"
thint.py:8: error: Argument 1 to "foo" has incompatible type "str"; expected
"int"
```

4 行目と 8 行目の整数値が期待されるところで、文字列が渡されているためにエラーが発生

エラーが発生したのは、先のコードで「# エラー」とコメントを入れた行と同じであり、静的型解析が機能していることが分かるはずだ。

ここまでは、Python 3.5 で追加された関数のパラメーター／戻り値型について型注釈を含めていたが、ここで先ほどのコードを次のように変更してみよう。

```
def foo(num: int) -> str:
  return str(num)

print(foo('string'))
print(foo(100))

num: int = '100'
print(foo(num))
```

変数にも型注釈を追加

Python 3.6 では「型注釈を変数に対しても付加できる」ようになったので、ここでは「変数 num は int 型」と注釈を付けている。これを Python 3.5 の mypy で静的に型解析すると、次のような結果になる（Python の環境構築の都合により、macOS 上で実行）。

```
$ python --version
Python 3.5.3

$ mypy thint.py
thint.py:7: error: Parse error before :

$ python thint.py
  File "thint.py", line 7
    num: int = '100'
       ^
SyntaxError: invalid syntax
```

Python 3.5 では変数に対する型注釈は行えない

→目次に戻る

ご覧の通り、mypy は 7 行目でパースエラーを発生している。その下の python コマンドでも構文エラーが発生した。これに対して、Python 3.6 で静的型チェックを行った様子が以下だ。

```
> python --version
Python 3.6.0 :: Anaconda 4.3.0 (64-bit)

> mypy --fast-parser --python-version 3.6 thint.py
thint.py:4: error: Argument 1 to "foo" has incompatible type "str"; expected
"int"
thint.py:7: error: Incompatible types in assignment (expression has type "str",
variable has type "int")

> python thint.py
string
100
100
```

Python 3.6 では変数に対する型注釈が可能

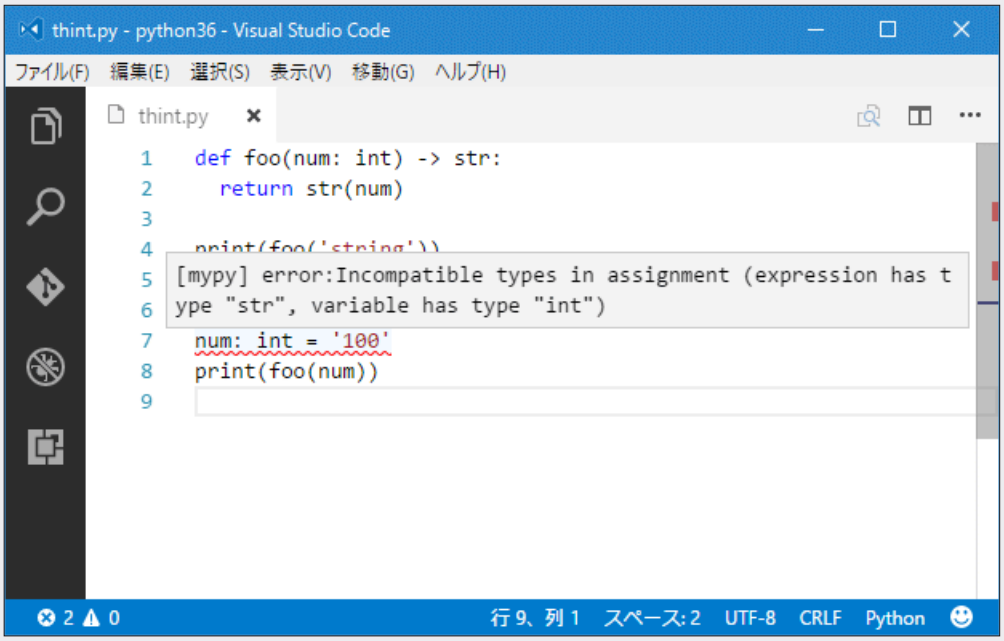
中央の出力に注目してほしい。まず、本項執筆時点では Python 3.6 の構文を mypy に解釈してもらうには「--fast-parser」オプションと「--python-version 3.6」オプションの指定が必要になる。また、前者のオプションを指定するには「typed_ast」パッケージの適切なバージョンも必要になる（そのため、上の pip コマンドではバージョンを指定して、typed_ast パッケージをインストールしている）。

出力結果を見ると、7 行目で「int 型の変数に文字列値を代入しようとした」ためにエラーが発生していることが分かる。Python 3.5 の mypy では「: num」の部分がパースエラーとなっていたが、Python 3.6 ではきちんとこれが解釈されているということだ。

さらに最後の python コマンドではエラーを発生することなく、スクリプトが実行できていることも分かる。

非常に簡単な例だが、Python 3.5 では関数のパラメーター／戻り値型に対する型注釈／型ヒント機能が実現され、Python 3.6 ではそれがさらに変数でも使用できるようになったことが分かったはずだ。Python の typing モジュールを使用すると、さらに高度な型指定や型操作も可能だが、これについては機会があれば別途ご紹介したい。

なお、mypy による型チェックが可能な環境では、Visual Studio Code に Python 拡張機能をインストールし、設定を記述することで、次のように Python コード記述時に静的型チェックを行うことも可能だ。



Visual Studio Code で Python コードの静的型チェック

最後に、非同期処理についても簡単に見ておこう。

非同期ジェネレータ／非同期内包表記

Python はバージョン 3.4 で非同期 I/O 処理を行うための asyncio パッケージが導入された。その後、バージョン 3.5 で async ／ await 構文による非同期イテレータを記述できるようになり（PEP 492）、バージョン 3.6 では async ／ await 構文による非同期ジェネレータがサポートされるようになった（PEP 525）。以下では、asyncio パッケージを使用して、バージョン 3.5 での非同期イテレータとバージョン 3.6 の非同期ジェネレータの利用について簡単に見ていこう。

まず「非同期」が付かないイテレータ（あるいはイテラブル=反復可能オブジェクト）とジェネレータについて簡単におさらいをしておく。Python のドキュメントではイテラブル（反復可能なオブジェクト）を「構成要素を一度に 1 つずつ返すことができるオブジェクト」と、イテレータを「データの流れを表現するオブジェクト」としている。

前者はリストなど Python に組み込みのデータ型や以下に示すような何らかの連続的なデータを格納し、それらを 1 つずつ取り出すことができるオブジェクトのことであり、後者は「現在、イテラブルに含まれているどの要素を扱っているかを認識し、要素を取り出したり、要素がなければそのことを（例外を通じて）通知したりできるオブジェクト」のことだ。実際にはイテラブルが「イテレータを取り出す __iter__ メソッド」と「次の要素を取得する __next__ メソッド」を提供し、イテレータとしても振る舞うことが多い。

単純なイテレータとジェネレータ

次に示すコードはそのように振る舞う単純なイテレータ（イテラブル）の例だ。

```
class MyIterable():
  def __init__(self, num):
    self.data = [x for x in range(num)]
    self.idx = -1

  def __next__(self):
    self.idx += 1
    if self.idx >= len(self.data):
      raise StopIteration
    else:
      return self.data[self.idx]

  def __iter__(self):
    return self

for item in MyIterable(3):
  print(item)
```

単純なイテラブルオブジェクト

__init__ メソッドでは num パラメーターの値に応じてデータを用意し、現在扱っている要素を示すインデックス値を初期化している（処理の都合でここでは「-1」としている）。__next__ メソッドは要素の取り出しを行うためのメソッドであり、for 文などでは内部的にこのメソッドが呼び出されている。ここでは、インデックス値をインクリメントした上で、まだデータがあればそれを返送し、なければ例外を発生している。__iter__ メソッドは反復処理が可能なオブジェクトが必要なところで自動的に呼び出されるメソッドであり、ここでは __next__ メソッドによる反復処理が可能である自分自身を返送している。

これに対して、ジェネレータは「イテレータを生成する関数」だと考えられる。以下に簡単な例を示す。これは上と同様な処理を行う「ジェネレータ - イテレータ」を生成する。

```
def MyGenerator(num):
  for item in range(num):
    yield item

for item in MyGenerator(3):
  print(item)
```

単純なジェネレータ

まず目に付くのはジェネレータは非常にコードがシンプルなことだ。また、全てのデータを最初に確保していないところも先ほどのイテレータとは異なる（この特性はメモリ負荷を削減することにもつながる）。次に、「return」で値を返すのではなく、「yield」で値を返しているところが異なる。もう 1 つ大きな違いは __next__ メソッドで

は繰り返しの中の一度の処理を記述していたが、ジェネレータが表現するのはデータを反復的に返送していく処理の全体を記述しているところだ。

ジェネレータでは「yield」により値を返送する際には、同時に制御も呼び出し側にいったん戻され、その後、ジェネレータが生成したイテレータに制御が移ると、以前に実行したコードの直後から実行が再開される（上のコードであれば、for 文で次のループが開始されるということだ）。必要な時点で必要な計算を行い、その結果を戻すといった場合にジェネレータは役に立つ。

こうした構造を模して、非同期に反復的な処理を行えるようにするための機構が asyncio パッケージと Python 3.5 ／ 3.6 で追加された async ／ await 処理ということになる。

非同期イテレータ

ただし、Python 3.5 でサポートされたのはあくまでも非同期イテレータまでだった。ちょっとサンプルとして難しいコードになってしまったが、非同期イテレータを使用したサンプルコードを以下に示す。これはコンソールで数値を入力してもらい、その値だけスリープした後にコンソールに出力を行い、再度入力を待つという動作をする（反復的に値を取り出す処理であるかどうかは難しいが、ユーザーが入力した値を返しているので許してほしい。また、エラー処理は省略している）。

```
import asyncio
import datetime

class MyAsyncIterable():
  def __aiter__(self):
    return self

  async def __anext__(self):
    time = input('input wait time (0 to break): ')
    time = int(time)
    if time == 0:
      raise StopAsyncIteration()
    await mysleep(time)
    return time

async def mysleep(time):
  to = datetime.datetime.now() + datetime.timedelta(seconds=time)
  while True:
    await asyncio.sleep(1)
    print('*', sep='', end='', flush=True)
    if datetime.datetime.now() > to:
      break
  print('')

async def main():
  aiter = MyAsyncIterable()
  async for msg in aiter:
    print('sleep time: {', msg, '} sec(s)', sep='')
  #running = True
  #while running:
  #  try:
  #    msg = await aiter.__anext__()
  #  except StopAsyncIteration as e:
  #    print('catch StopAsyncIteration')
  #    running = False
  #  else:
  #    print(f'sleep time: {msg} sec(s)')

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

非同期イテレータの利用例（gen35.py ファイル）

ここで注目してほしいのは、MyAsyncIterable クラスと main 関数だ。mysleep 関数については「1 秒ごとにアスタリスクをコンソールに出力しながら、指定された秒数だけ待機する関数」だと思ってほしい（詳しい説明は割愛）。また、コード末尾の 3 行は非同期 I/O を行う際の定型処理だ。イベントループを取得して、そこに実行したい処理を投げ込んで、最後にループをクローズしている。

そこでまずは MyAsyncIterable クラスを見てみよう。

```
class MyAsyncIterable():
  def __aiter__(self):
    return self

  async def __anext__(self):
    time = input('input wait time (0 to break): ')
    time = int(time)
    if time == 0:
      raise StopAsyncIteration()
    await mysleep(time)
    return time
```

MyAsyncIterable クラスの構造は MyIterable クラスと似ている

最初に見たシンプルな MyIterable クラスでは __init__ メソッドでデータの初期化などを行っていたが、ここではデータを保持していないので __init__ メソッドはない。あるのは __aiter__ メソッドと __anext__ メソッドの 2 つだ。前者の名前は「__iter__」に、後者の名前は「__next__」に非同期（asynchronous）を意味する「a」をそれぞれ付加したものだ。

やっていることも同様であり、__aiter__ メソッドはイテレータとして自身を返送している（__aiter__ メソッドの仕様は Python 3.5.2 で変更されているため、このコードはそれより前のバージョンでは動作しない。詳細は「PEP 492 - コルーチン、async と await 構文」の「注釈」を参照されたい）。また、__anext__ メソッドでは「反復処理の中の 1 回の処理」を記述している。

注意したいのは、関数定義が「async def」で始まっているところと、内部で await による処理の待機（「await mysleep(time)」で指定された秒数だけ待機する関数呼び出しが終了するのを待機している）を行っているところだ。「await」は「async def」で定義される関数内でのみ使用可能であり、他の部分では構文エラーとなる。待機秒数として「0」が入力された場合には「StopAsyncIteration」例外を送出しているが、これは main 関数での「async for」文で反復処理の終了を判断するために使われている）。

main 関数は次のようになっている。

```
async def main():
  aiter = MyAsyncIterable()
  async for msg in aiter:
    print('sleep time: {', msg, '} sec(s)', sep='')
  #running = True
  #while running:
  #  try:
  #    msg = await aiter.__anext__()
  #  except StopAsyncIteration as e:
  #    print('catch StopAsyncIteration')
  #    running = False
  #  else:
  #    print(f'sleep time: {msg} sec(s)')
```

main 関数

main 関数では「async for」文でループ処理を行っている。非同期イテレータ／ジェネレータを利用して反復処理を行うには通常の for 文ではなく、async for 文を使用する必要がある。また、「await」と同じく、これを async for 文は async def で定義される関数内でしか使えないことにも注意しよう。

コメントアウトしてあるのは、その上の async for ループと同等な処理だ。async for ループと下のコードでコメントイン／コメントアウトを入れ替えて実行してみると、StopAsyncIteration が発生した時点でループが終了するのが分かるはずだ。

実際に実行した様子を以下に示す。

```
> python gen35.py
input wait time (0 to break): 3
***
sleep time: {3} sec(s)
input wait time (0 to break): 0

> python gen35.py
input wait time (0 to break): 1
*
sleep time: 1 sec(s)
input wait time (0 to break): 0
catch StopAsyncIteration
```

実行結果

上は async for ループを利用した場合で、下は while ループを使用した場合のもの。

非同期イテレータの特性をまとめると次のようになる。

- __aiter__ メソッドを持つ
- __anext__ メソッドを持つ
- 非同期に行われる処理を待機するには await キーワードを使用
- await できるのは async def で定義された関数内のみ
- 非同期に反復処理を行うには async for 文を使用
- async for 文を記述できるのは async def で定義された関数内のみ

そして、非同期ジェネレータはこうした特性のうち、__aiter__ メソッド／ __anext__ メソッドの定義が必要になる非同期イテレータの実装を隠してくれる便利なジェネレータだといえる。

非同期ジェネレータ

上で見た非同期イテレータと同等な処理を行う非同期ジェネレータのコードを以下に示す。

```
import asyncio
import datetime

async def MyAsyncGenerator():
  while True:
    time = input('input wait time (0 to break): ')
    time = int(time)
    if time == 0:
      break
    await mysleep(time)
    yield time

async def mysleep(time):
  # …… 省略 ……

async def main():
  agen = MyAsyncGenerator()
  async for msg in agen:
    print(f'sleep time: {msg} sec(s)')
  #running = True
  #while running:
  #  try:
  #    msg = await agen.__anext__()
  #  except StopAsyncIteration as e:
  #    print('catch StopAsyncIteration')
  #    running = False
  #  else:
  #    print(f'sleep time: {msg} sec(s)')

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

非同期ジェネレータ

→目次に戻る

先ほどはクラスとして MyAsyncIterable を定義していたが、MyAsyncGenerator は非同期ジェネレータ関数として「async def」を利用して定義している。単純なジェネレータ関数と同様、反復処理の全体をまとめて記述している点に注意しよう（ここでは MyAsyncIterable.__anext__ メソッドと同じ処理を「while True:」による無限ループにくるんでいる）。内部での処理は MyAsyncIterable.__anext__ メソッドとほぼ同じだが、「return」ではなく「yield」で値（と制御）を戻している点と、ユーザーが「0」を入力し、反復処理を終了する際に StopAsyncIteration 例外を送出していないのが異なる。非同期ジェネレータでは StopAsyncIteration 例外がラップされるので、自前でこの例外を送出する必要はない。

main 関数での処理は変数名、クラスのインスタンス生成／ジェネレータ関数呼び出しが異なるだけでやっていることは同じだ。コメントアウトの内容も同等なので、main 関数で行う処理を入れ替えて実行し、本当に StopAsyncIteration 例外が発生しているかを確認してみてほしい。

Python 3.x での非同期処理に関しては、他にも待機可能オブジェクトや asyncio パッケージ関連の話題など、語るべきことが多数あるが別項に譲ることにする。

最後に非同期ジェネレータに関連して、非同期処理を内包表記で使用する簡単なサンプルも紹介しておこう（PEP 530）。これは 2 秒待機した後に、数値を返送するジェネレータとそれを利用するコードだ。コードは見れば分かる通りなので、説明は割愛する。

```
import asyncio
from random import randint

async def SimpleAsyncGenerator(num):
  for item in range(num):
    await asyncio.sleep(2)
    yield item

async def main():
  result = [x async for x in SimpleAsyncGenerator(3)]
  print(result)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

非同期内包表記

◇ ◇ ◇ ◇ ◇ ◇

本章では、Python 3.6 で追加された機能の中から、文字列補間（f 文字列）、数値リテラル内でのアンダースコアの利用、拡張された型注釈機能、非同期プログラミングについて紹介した。機会があれば、ここで取り上げられなかったものもまた紹介していこう。

@ IT eBook シリーズ　Vol.33
機械学習で注目のPythonを学ぶVisual Studioで始めるPythonプログラミング入門
制作：アイティメディア @ IT 編集部
2017 年 10 月 26 日