ご利用のコンピュータを設定する方法

このラボの作業を行うには、事前設定された dCloud ラボを使用するか、自身のコンピュータをセットアップします。詳細については、イベントの事前準備 [英語] とラボの設定 [英語] の両方のモジュールを参照してください。

Python によるコーディングと JSON の解析

このラーニング ラボでは、Python の構文と、Python で JSON を解析する方法について詳しく学習します。

目標

所要時間：45 分

- Python でのコーディングの基礎を理解する
- Python を使用して、シンプルな形式の JSON からより複雑な形式の JSON までを解析する方法について学習する

前提条件

背景

- このラボを開始する前に、『Coding 101 Rest Basics Learning Lab（コーディング 101：Rest の基本に関するラーニング ラボ）』[英語] と Coding 102 Rest Python Learning Lab（コーディング 102：Rest Python に関するラーニング ラボ）[英語] を完了しておくことをお勧めします。

Python

- サンプル コードを実行するためには、使用するマシンに Python 3 がインストールされている必要があります。
- **DevNet イベントで DevNet ラーニング ラボ PC を使用している**場合は、Python 3.x + がすでにインストールされています。
- Python をインストールする方法については、この ページ上部の「**ご利用のコンピュータを設定する方法**」の項を参照してください。

Python の requests ライブラリ

- いくつかのサンプル コードでは、Python の requests ライブラリを使用して REST API の呼び出し部分を簡単に行えるようにしています。

- **DevNet イベントで DevNet ラーニング ラボ PC を使用している場合は、**requests ライブラリがすでにインストールされています。
- requests ライブラリをインストールする方法については、この ページ上部の「**How to Setup Your Own Computer（ご利用のコンピュータを設定する方法）**」の項を参照してください。

Git Repo を複製する

- **DevNet イベントで DevNet ラーニング ラボ PC を使用している場合は、**
  - タスク バー上の *Git CMD* アイコンをクリックして Git コマンド ウィンドウを開くか、またはスタート ボタンをクリックして、実行バーに「**git cmd**」と入力して **Enter** キーを押してください。
- **自分のワークステーションで作業を行っている場合は、**デスクトップでコマンド ターミナルを開きます。
- `cd \` と入力して、ルート ディレクトリに移動します。
- `mkdir DevNetCode\<your-name>` と入力して、「C:\DevNetCode\yourname」というディレクトリを作成します。
  - 例：mkdir DevNetCode\brTiller
- `cd \DevNetCode\<your-name>` と入力して、新しいディレクトリに移動します。
  - 例：`cd \DevNetCode\brTiller`
- GitHub から devnet-express-code-samples リポジトリを複製します。次のコマンドを入力します。
- `git clone https://github.com/CiscoDevNet/devnet-express-code-samples`

```
C:\DevNetCode\brTiller>git clone https://github.com/CiscoDevNet/devnet-express-code-samples.git
Cloning into 'devnet-express-code-samples'...
remote: Counting objects: 148, done.
remote: Total 148 (delta 0), reused 0 (delta 0), pack-reused 148Receiving objects:  60% (89/148)
Receiving objects: 100% (148/148), 41.09 KiB | 0 bytes/s, done.
Resolving deltas: 100% (78/78), done.
Checking connectivity... done.
```

作成したディレクトリに「devnet-express-code-samples」ディレクトリが確認できるはずです。'

```
C:\DevNetCode\brTiller>dir
 Volume in drive C is System
 Volume Serial Number is 808B-CEEA

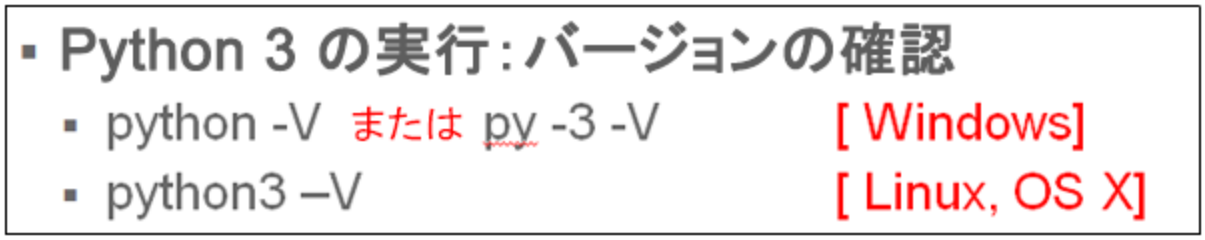
 Directory of C:\DevNetCode\brTiller

12/15/2015  03:16 PM    <DIR>          .
12/15/2015  03:16 PM    <DIR>          ..
12/15/2015  03:16 PM    <DIR>          devnet-express-code-samples
               0 File(s)              0 bytes
               3 Dir(s)  55,545,159,680 bytes free
```

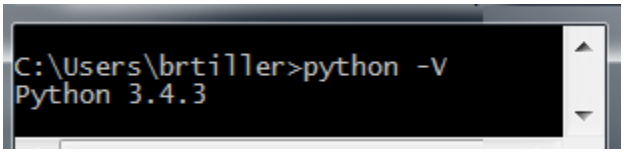
ステップ 1：Python のバージョン、スクリプト、対話型シェル

Python のバージョンを確認

システムにインストールされている Python のバージョンは簡単に確認できます。ターミナル ウィンドウにアクセスし、コマンド プロンプトに「**python -V**」と入力して、リターン キーを押します。複数のバージョンの Python または Python 3 がインストールされている場合は、オペレーティング システムに応じて「**py -3 -V**」または「**python3 -V**」と入力します。詳細については、下の図を参照してください。

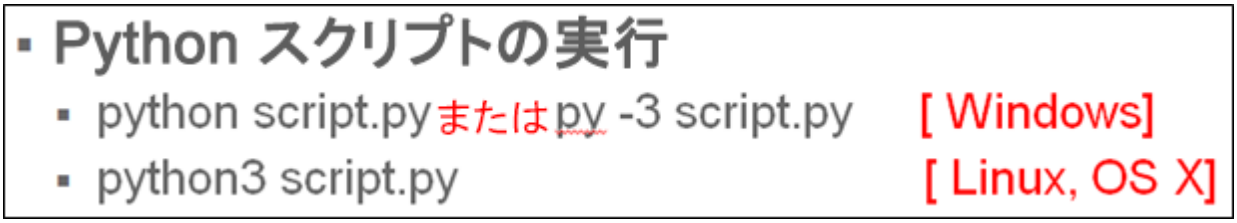


Python のバージョンを確認する例を示します。

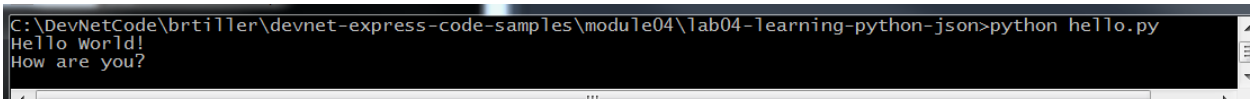


Python スクリプトの実行

Python スクリプトは簡単に実行できます。ターミナル ウィンドウのコマンド プロンプトに「**python**」と入力し、Python スクリプトのフルネームを入力してリターン キーを押します。Python スクリプトが保存されているディレクトリ以外の場所では、スクリプトのあるディレクトリの場所をすべて正確に指定する必要があります。



Python スクリプトを実行する例を示します。

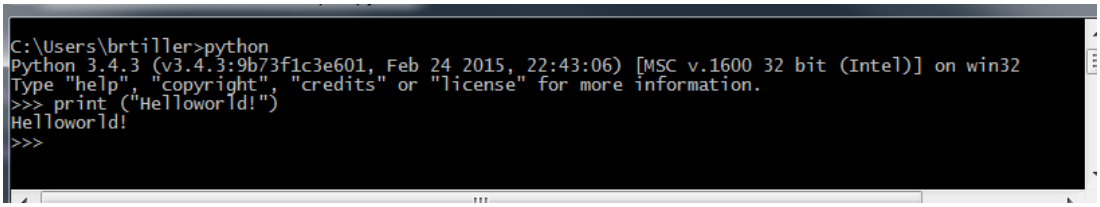


Python の対話型シェルを利用してインタプリタを実行

コマンド プロンプトで「**python**」と入力してリターン キーを押すだけで、Python の対話型シェルを開くことができます。対話型シェルでは、非常に基本的な Python コードを記述して、対話形式でインタプリタを実行できます。終了するには、Ctrl+Z を押すか「quit()」と入力してリターンキーを押します。



Python のインタプリタを対話型シェルで使用する例を示します。



実際にやってみよう

Python の対話型シェルを使ってみる

1. ターミナルを開きます。
2. 利用しているオペレーティング システムに応じて適切な Python コマンドを入力し、バージョンを確認します。たとえば Windows の場合は、「**python -V**」と入力してリターン キーを押します。
3. 適切な Python コマンドを入力して、対話型シェルを開始します。たとえば Windows の場合は、「**python**」と入力してリターン キーを押します。
4. 「**print("Hello World! How are you?")**」と入力してリターン キーを押します。
5. 終了するには **Ctrl** キーと **z** を押すか、「**quit()**」と入力します。

Python スクリプトを実行してみよう

1. ターミナルを開き、先に作成したディレクトリ **DevNetCode\<your-name>** に移動します。
    - 「**前提条件**」セクションでは、ソース コード ファイルを Git リポジトリからこのディレクトリに複製しました。作成したディレクトリに **devnet-express-code-samples** サブディレクトリが確認できるはずです。このサブディレクトリがない場合は、前提条件のセクションに戻り、手順に従ってディレクトリを作成し、Git リポジトリを複製します。
2. **module04\lab04-learning-python-json** ディレクトリに移動します。ターミナルに「**cd \DevNetCode\<your-name>\devnet-express-code-samples\module04\04-lab04-learning-python-json**」と入力します。
3. スクリプトを実行するには、Python コマンドを入力後、コマンド プロンプトにファイル名を入力して、リターン キーを押します。
    - *Windows では*「**py -3 hello.py**」と入力します。または「**python hello.py**」と入力します。
    - *Mac OS または Linux では*「**python3 hello.py**」と入力します。
4. プログラムが実行されるか、エラー メッセージが表示されます。

次のステップ : Python のスコープ、変数、演算子、単純な条件ステートメントについて学習します。

ステップ 2：Python – スコープ、変数、演算子、単純な条件ステートメント

このステップでは、スコープ、変数、そして条件ステートメントの基本事項を確認します。

Python の基本

他の言語でコードを記述したことがあれば、スコープ（ステートメントの影響範囲または、変数が存在する有効範囲）を定義するステートメント ブロックが波括弧を使って指定されているのをよくみるでしょう。これらの括弧は、ステートメントの開始と終了、つまり範囲の定義に使用されます。Python では、スコープの定義に波括弧は使用しません。代わりに、インデントを使用します。インデントの形式としては、スペースまたはタブを使用しますが、どちらかを一貫して使用する必要があります。一貫していないと Python のインタプリタでエラーが発生します。

次の Python コードを見てください。

```
print ("Hello World!")
num = 1
if num < 1:
   print ("I'm less than 1!")
   print ("Goodbye Cruel World!")
```

1. 1 行目では、画面に「Hello World!」というテキストを表示させます。これは開発者が初めてアプリケーションを記述する際に、一般的に書かれるものです。
2. 2 行目では、「num」という変数を作成し、値として 1 を割り当てています。この変数の名前は「brett」でも「a」でも問題ありません。詳細については、変数の名前付けに関するページ [英語] を参照してください。
3. 次の行では、「num」という名前を付けた変数の値が 1 未満であるかどうかを確認します。演算子には「<」（より小さい）を使用します。Python には多様な演算子が用意されていますが、最も一般的に使用するのは比較、割り当て、算術演算子です。この if ステートメントは、条件を確認するための *条件ステートメント* と呼ばれます。条件が True であれば、条件ステートメントのスコープ内にあるすべてが実行され、True でない場合はすべてスキップされます。
4. 次の 2 つのステートメントはインデントされています。このインデントは、条件ステートメントの範囲内にあることを示しています。

クイズ:解答は下を参照

1. 上記の Python コードを実行すると、画面に何が出力されるでしょうか。
2. print("Goodbye Cruel World!") の行をインデントしない場合、画面に何が出力されるでしょうか。
3. num の値を 0 に変更すると、画面に何が出力されるでしょうか。

解答

1. Hello World!
2. Hello World! Goodbye Cruel World!
3. Hello World! I'm less than 1! Goodbye Cruel World!

実際にやってみよう

1. ターミナルを開き、ステップ 1 で作成した **DevNetCode\<your-name>** という名前のディレクトリに移動します。
    - ステップ 1 の「**前提条件**」の項では、ソース コード ファイルを Git リポジトリからこのディレクトリに複製しました。作成したディレクトリに **devnet-express-code-samples** サブディレクトリが確認できるはずです。このサブディレクトリがない場合は、ステップ 1 の前提条件の項に戻り、手順に従ってディレクトリを作成し、Git リポジトリを複製します。
2. **module04\lab04-learning-python-json** ディレクトリに移動します。ターミナルに「**cd \DevNetCode\<your-name>\devnet-express-code-samples\module04\04-lab04-learning-python-json**」と入力します。
3. スクリプトを実行するには、Python コマンドを入力後、コマンド プロンプトにファイル名を入力して、リターン キーを押します。
    - *Windows では*「**py -3 helloworld.py**」と入力します。または「**python helloworld.py**」と入力します。
    - *Mac OS または Linux では*「**python3 helloworld.py**」と入力します。
4. プログラムが実行されるか、エラー メッセージが表示されます。

以下の変更を行い、helloworld.py スクリプトを実行します。

1. **helloworld.py** ファイルを開きます。たとえば Windows では、「**notepad helloworld.py**」と入力します。
2. クイズの問題 2 で示された変更を行います。
3. ファイルを保存します。エンコーディング タイプがオプションの場合は、**UTF-8** を選択します。
4. スクリプトを実行します。
5. ステップ 1 ～ 4 を繰り返し、クイズの問題 3 で示された変更を行います。

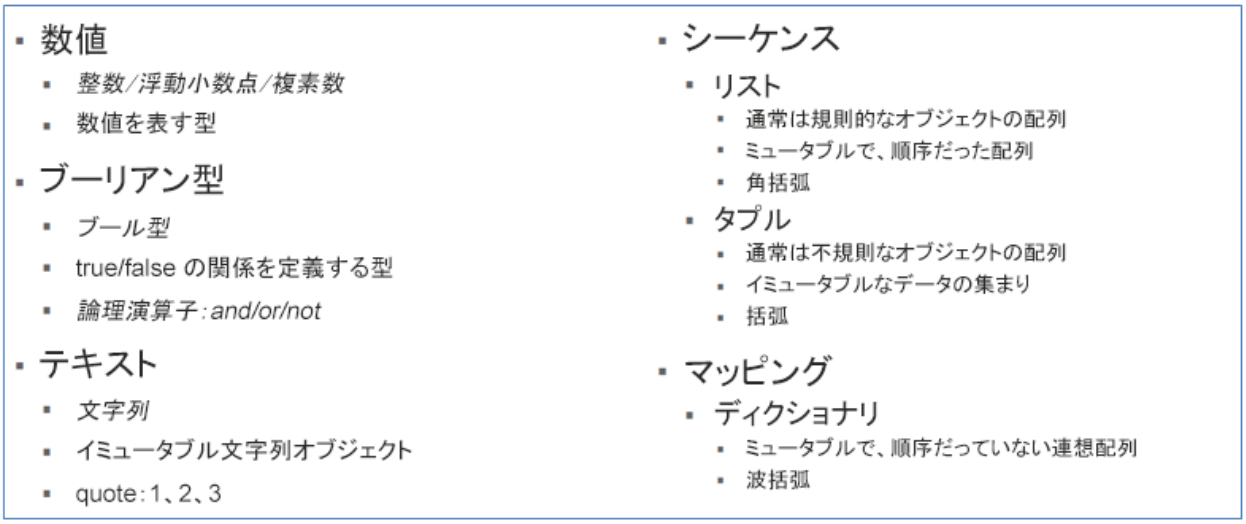
初めてのプログラムを作成する

1. myworld.py というファイルを作成します。
2. 次のことを実行する Python コードを記述します。
   a. 変数に値を割り当てる
   b. 変数の値を確認する　ヒント:「if (variable < value):」という書き方を思い出そう
   c. 条件が true である場合、*if* ステートメントのスコープの実行内容を出力する。if 条件ステートメントのスコープで実行したい命令はインデントする必要があります。

次のステップ: Python のデータ型について学習します。

ステップ 3：Python のデータ型

このステップでは Python のデータ型について学習します。次の表に複数のデータ型の一覧を示しています。**整数**や**浮動小数点**など、シンプルな数値のデータ型もあります。**浮動小数点**と**整数**の違いは、**浮動小数点**には小数が含まれていることです。たとえば 52.856 は**浮動小数点**で、52 は**整数**です。**Boolean** のデータ型は、True または False をとります。**テキスト**は、数値以外の値を持つデータ型です。**テキスト**は一般的に**文字列**と呼びます。たとえば「brett」は**文字列**であり、「brett123」も文字列ですが、「123」は**整数**です。



リスト、タプル、ディクショナリなど、さらに複雑なデータ型もあります。これらのデータ型について詳しく見ていきましょう。

リストには、任意の数の連続する要素が含まれます。角括弧 [] によって定義します。*['Martha','Betty',5]* というリストがあるとします。変数にリストを割り当てるには、単純に *var=['Martha','Betty',5]* のようにします。リストはミュータブルです。つまり、変更することが可能です。リストにはデータを追加でき、またデータの変更や削除、ソートが可能です。リストには異なるデータ型を含めることができます。

タプルには、任意の数の連続する要素が含まれます。括弧 () によって定義します。*('Brett',9,'Cisco')* というタプルがあるとします。変数にタプルを割り当てるには、単純に *var1=('Brett',9,'Cisco')* のようにします。タプルはリストに似ていますがイミュータブルです。つまり、変更できません。タプルは、関連するデータをグループ化する場合などに使用します。たとえば、上記のタプルでは、名前が Brett で、シスコに 9 年間勤務しているという私についての関連性を定義しています。タプルはイミュータブルであるため、リストよりもすばやくデータにアクセスすることができるという点で効率的です。

リストとタプルは定義が多少異なりますが、どちらも角括弧 [] を使用してデータにアクセスすることができます。1 つ目のデータ要素は常に位置ゼロから開始されます。先に定義したリストの例では、Python ステートメント **print(var[0])** を入力すると、*Martha* と表示されます。Python ステートメント **print(var[1])** を入力すると、*Betty* と表示されます。以下同様です。同じ構文をタプルにも適用できます。上で定義したタプルでは、Python ステートメント **print(var1[0])** を入力すると、*Brett* と表示されます。Python ステートメント **print(var1[1])** を入力すると、*9* と表示されます。以下同様です。

ディクショナリは、リストやタプルとは異なります。ディクショナリ内の各要素には、1 つのキーに続いて値が必要になります。このキーと値の関連付けは、一般に*名前/値*ペアと呼ばれます。ディクショナリは波括弧 {} で定義します。ディクショナリ *{"car":"corvette","age":7, "food":"pizza"}* があるとします。変数にディクショナリを割り当てるには、単純に *myvar={"car":"corvette","age":7, "food":"pizza"}* のようにします。ディクショナリ内の値には、キーを使用してアクセスできます。たとえば Python ステートメント **print(myvar["car"])** を入力すると、*corvette* と表示されます。Python ステートメント **print(myvar["food"])** を入力すると、*pizza* と表示されます。以下同様です。ディクショナリは、マップまたは*連想配列*と呼ばれる場合があります。

次の表に、Python の各データ型の定義例を示します。

タイプ	例
• 数値：整数、浮動小数点	x = 10 x = 1.0
• 文字列型（String）	x= 'Mike'
• ブーリアン型（Boolean）	y = True x = False
• リスト	my_list = [10, 20, 30]
• タプル	my_tuple = ('Brett', 'Cisco', 'Cary',2015)
• ディクショナリ	my_dict = {"one":1, "two":2}
• リスト内リスト	my_list2=[[10,20,30], ['Cisco Live', 'May', 2016]]

実際にやってみよう

1. ステップ 2 では、Python スクリプト myworld.py を作成して実行しました。このスクリプトに次のように追加します。
   a. 2 つの要素を含むリストを追加し、変数に割り当てます。
   b. 3 つの要素を含むタプルを追加し、変数に割り当てます。

c. 2 つの名前/値ペアを含むディクショナリを追加し、変数に割り当てます。
d. 作成した各データ型から、1 つの値を出力します。

2. 次に data-types-values.py ファイルを開きます。各データ型について、どのような値が表示されるでしょうか。スクリプトを実行して確認してみましょう。
3. data-type-values.py ファイルを変更して、エラーを発生させます。たとえば、*print(my_dict["green"])* を *print(my_dict["mean"])* に変更します。どのようなエラーが表示されるか見てみましょう。

次のステップ : JSON について、また Python を使用して JSON を解析する方法について学習します。

ステップ 4：Python を使用して JSON を解析する

Python のデータ型に関する知識に基づいて、JSON について、また Python を使用して JSON を解析する方法について学習します。JSON（**Java Script Object Notation**）は、テキスト ベースの名前/値ペアで構成されており、アプリケーションでのデータの保存と交換が容易になります。軽量で読みやすく、最低限の内容で設計されているため、データの交換と保存に使用されるもう 1 つのテキスト ベース プロトコルである XML でみられるような、過剰なテキストという問題が起こりません。

JSON の構造と解析は、Python のディクショナリとリストに非常に似ています。

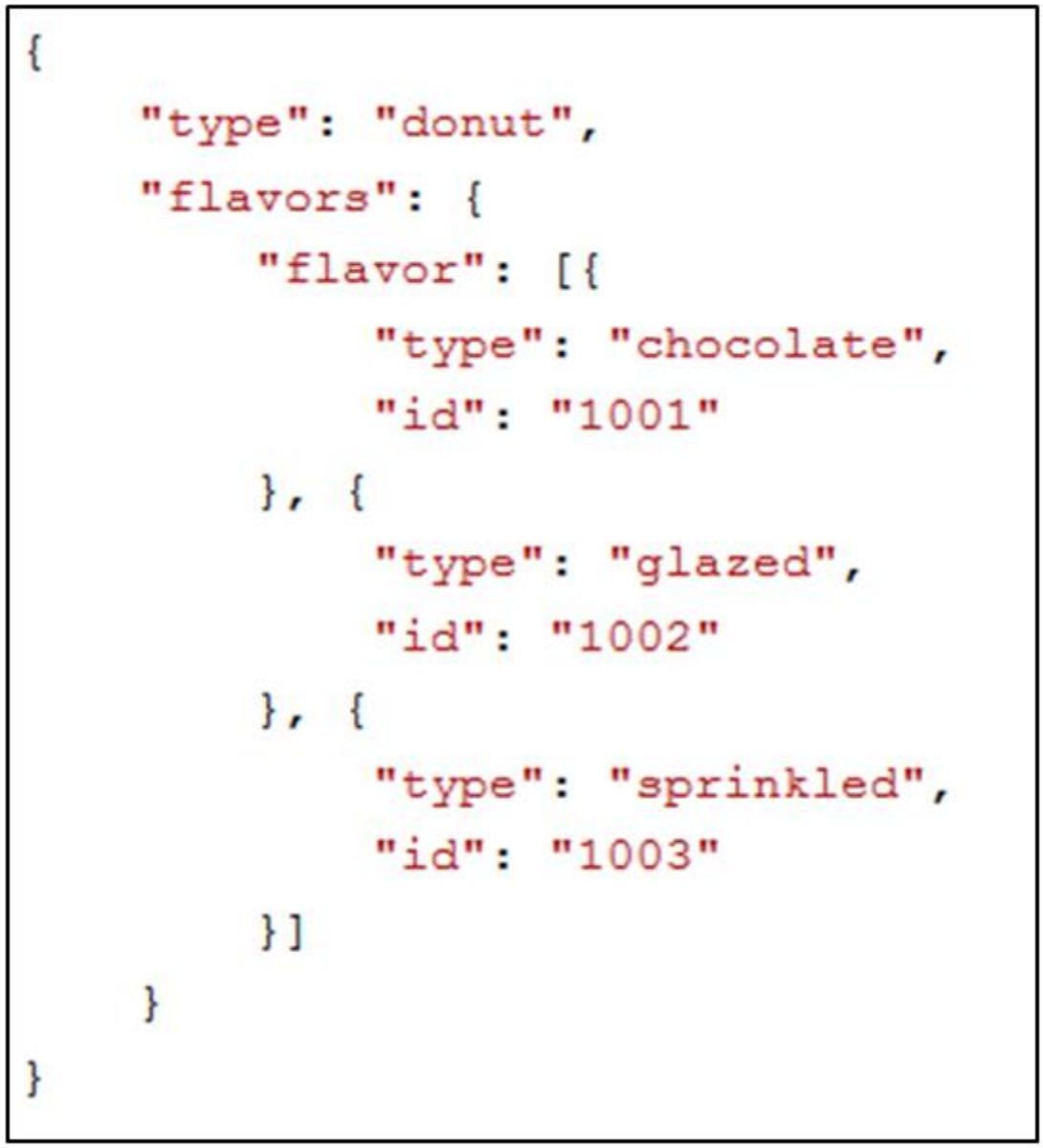
1. JSON データは、Python ディクショナリと同様に名前と値のペアで設定されます。ただし JSON の**ディクショナリ**は**オブジェクト**と呼ばれます。たとえば、**{"car":"volvo", "fruit":"apple"}** は JSON オブジェクトです(見た目は Python のディクショナリと同じことにお気づきだろうか)。多くの場合、Python では、JSON オブジェクトを、Python のディクショナリとして表現することができます。JSON オブジェクトを Python のディクショナリとして表現しているので、これまで学んだように変数に割り当てることもできます（**var={"car":"volvo", "fruit":"apple"}**）。さらに、データへのアクセス方法も、ディクショナリ内のデータへのアクセスと同じになります。たとえば **fruit** の値を得るには、「**var["fruit"]**」と入力します。**apple** が返されます。この値を表示するには「**print(var["fruit"])**」と入力します。

2. Python と同様に、JSON でも**リスト**を**配列**として使用します。JSON では、配列は一般的にオブジェクト内にネストされています。たとえば **{"donut":["chocolate","glazed","sprinkled"]}** のようになります。この JSON オブジェクトをディクショナリとして変数に割り当てると、次のようになります。**var1={"donut":["chocolate","glazed","sprinkled"]}**ここでは **donut** が**キー**であり、**値**がドーナツの種類を示す配列になっています。チョコレートドーナツが欲しい場合は（そろそろお腹もすいたでしょう）、「**var1["donut"] [0]**」と入力します。配列内の最初の要素である **chocolate** が返されます。テキストを表示するには、「**print(var1["donut"] [0])**」と入力します。

3. JSON では、オブジェクトの中に、配列を持ったオブジェクトをネストすることもできます。例：**{"donut":{"flavors":["chocolate","glazed","sprinkled"]}}**。このネストでは、**donut** が**キー**で、オブジェクト **flavors** がその値になっています。ただし、**flavors** も同様に**キー**であり、その値がドーナツの種類を示す配列になっています。そのためチョコレートドーナツを得るには、さらに処理が必要です。まず JSON オブジェクトを表すディクショナリを変数に割り当てます。
myvar={"donut":{"flavors":["chocolate","glazed","sprinkled"]}}
chocolate にアクセスするには、「**myvar["donut"]["flavors"][0]**」と入力します。これを表示するには「**print(myvar["donut"]["flavors"][0])**」と入力します。一般的に JSON は、データタイプごとにインデントを入れてフォーマットすることで読みやすくすることができます。上記の例は、次の図のようにフォーマットすることができますが、データの解析は同様の方法で行われます。

```
{
    "donut": {
        "flavors": ["chocolate", "glazed", "sprinkled"]
    }
}
```

4. また、JSON では、リスト内にオブジェクトを含むことがあります。例：
{"type":"donut","flavors":{"flavor":[{"type":"chocolate","id":"1001"},
{"type":"glazed","id":"1002"},{"type":"sprinkled","id":"1003"}]}}。どの要素がキー、値、またはキーと値の両方であるかを確認してみてください。ただしチョコレートドーナツを得るには、さらに処理を進める必要があります。まずこれを変数に割り当てます。
myvar1={"type":"donut","flavors":{"flavor":[{"type":"chocolate","id":"1001"},
{"type":"glazed","id":"1002"},{"type":"sprinkled","id":"1003"}]}}
chocolate にアクセスするには、「**myvar1["flavors"]["flavor"][0]["type"]**」と入力します。これを表示するには「**print(myvar1["flavors"]["flavor"][0]["type"])**」と入力します。先に説明したように、JSON は一般的に読みやすくなるようなフォーマットで記述することができます。上記の例は通常は次の図のようになります、データの解析は同じ方法で行われます。



```
{
    "type": "donut",
    "flavors": {
        "flavor": [{
            "type": "chocolate",
            "id": "1001"
        }, {
            "type": "glazed",
            "id": "1002"
        }, {
            "type": "sprinkled",
            "id": "1003"
        }]
    }
}
```

クイズ:解答は下を参照

fruit={"cherry":"red", "lemon":"yellow", "lime":"greeen"}

food={“vegetables”:[“carrots”,“kale”,“cucumber”,”tomato”]}

cars={“sports”:{“Porsche”:“Volkswagon”,“Viper”:“Dodge”,“Corvette”:“Chevy”}}

donut={"type":"donut","flavors":{"flavor":[{"type":"chocolate","id":"1001"},
{"type":"glazed","id":"1002"},{"type":"sprinkled","id":"1003"}]}}

次のそれぞれに求められる内容にアクセスし、表示するコードを記述してください。

1. fruit 変数から「lemon」の値にアクセスする
2. food 変数から「tomato」にアクセスする
3. cars 変数から「Viper」の値にアクセスする
4. donut 変数から「glazed」の値にアクセスする

解答

1. fruit["lemon"] print(fruit["lemon"])
2. food["vegetables"][3] print(food["vegetables"][3])
3. cars["sports"]["Viper"] print(cars["sports"]["Viper"])
4. donut["flavors"]["flavor"][1]["type"] print(donut["flavors"]["flavor"][1]["type"])

実際にやってみよう

1. ディレクトリ \DevNetCode\<your-name>\devnet-express-code-samples\module04\04-lab04-learning-python-json に移動します。
2. **json_parse-1.py** ファイルを開きます。
3. Python のディクショナリで表現された各 JSON オブジェクトに対して 1 つの値を出力する Python コードを記述します。コードを実行します。

次のステップ：Python でループを記述する方法を学習します。

ステップ 5:ループ

このステップでは、Python でループと関数を記述する方法を学習します。ここでは、この新しいスキルを使用して、配列、タプル、ディクショナリ内に複数の要素を含む JSON データを簡単に解析できるようにします。

Python またはその他のプログラミング言語でループを記述する場合、そこにはさまざまな理由があります。最も一般的な理由は、リストを処理する必要があるというものです。前のステップでは、JSON 配列内のオブジェクトにアクセスする場合に、要素番号を指定する必要がありました。しかし、オブジェクトが何百、何千とある場合には、この作業には非常に手間がかかります。各オブジェクトでループが反復されることで、記述したソース コードを使用した処理が可能となり、リストの処理が大幅に簡単になります。

Python で使用する各種のループを見てみましょう。

1 つ目のタイプのループでは、range 関数を使用して、ループ内で実行される反復の回数を固定値で指定します。最初の例では、5 回の反復が指定されています。Python のデフォルトではカウントが 0 から開始されるため、0 から 4 で 5 回の反復が行われることになります。次の例では、開始点と終了点を指定します。Python では最後の数字が除外されるため、反復は 2、3、4 の 3 回になります。なお、**in** キーワードは重要です。このキーワードにより、範囲内の値がカウント用の変数に繰り返し割り当てられます。つまり今回の例では、変数名 **count** が反復のたびに 1 つずつ増加していきます。

同様に **for fruit in basket** の例の場合は、**basket** 配列が繰り返し処理され、反復のたびにその値が **fruit** という名前の変数に割り当てられた上で、解析されます。最後の 2 つの例の **while ループ**では、変数に値が割り当てられません。一般的には、入力の処理やリストではないデータの処理に使われます。たとえば **while True** は無限ループを意味し、ユーザの入力を待機する場合に使用されることがあります。

for count in range(5)	0 ～ 4 のループ
for count in range(2,5)	2 ～ 4 のループ
for fruit in basket	For 変数 in リスト形式のステートメント
while count < 5	カウントのインクリメントが必要
while True	無限ループ。ブレイク ステートメントで終了

以上のようなループに関する新しい情報をもとに、さらに多くの JSON データをより効率的に処理できることを確認しましょう。ステップ 4 では、ドーナツに関する JSON データを解析し、*chocolate* の値を取得しました。ただ、すべての種類のドーナツとそれに対応する ID を、ループを使用せずに取得しようとすると、非常に手間と時間がかかります。次の例を参照してください。

```
donut={"type":"donut","flavors":{"flavor":[{"type":"chocolate","id":"1001"},
{"type":"glazed","id":"1002"},{"type":"sprinkled","id":"1003"}]}}

# JSON をループを使わずに解析する
print(donut["flavors"]["flavor"][0]["id"] + "  " +
donut["flavors"]["flavor"][0]["type"])
print(donut["flavors"]["flavor"][1]["id"] + "  " +
donut["flavors"]["flavor"][1]["type"])
print(donut["flavors"]["flavor"][2]["id"] + "  " +
donut["flavors"]["flavor"][2]["type"])
print()

# JSON をループを使って解析する
for hungry in donut["flavors"]["flavor"]:
    print(hungry["id"] + "  " + hungry["type"])
```

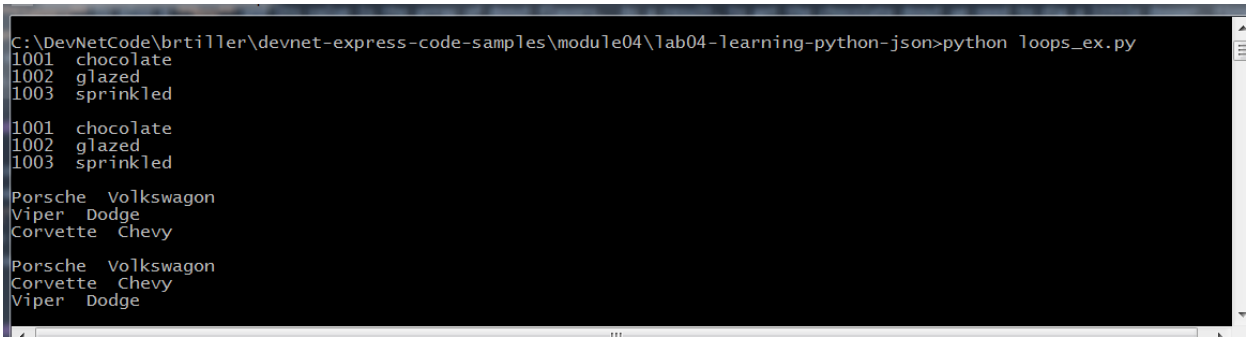
ループを使用して、配列ではないデータ構造(ディクショナリなど)の解析をすることもできます。たとえばステップ 4 では、自動車に関する JSON データを解析しました。しかし、すべての自動車の型と対応するモデルを出力する場合はどうでしょうか。ループを使用すれば、それぞれを問題なく出力することができます。次に、従来の方法と改善された方法を比較してみます。

```
cars={"sports":{"Porsche":"Volkswagon","Viper":"Dodge","Corvette":"Chevy"}}

# JSON をループを使わずに解析する
print("Porsche  " + cars["sports"]["Porsche"])
print("Viper  " + cars["sports"]["Viper"])
print("Corvette  " + cars["sports"]["Corvette"])
print()

# JSON をループを使って解析する
for auto in cars["sports"]:
    print(auto + "  " + cars["sports"][auto])
```

以下は、従来の方法を使用したスクリプトと新しい方法を使用したスクリプトのそれぞれで JSON データの解析を実行した際の出力結果です。



これまでに学習したことを確認してみましょう。以下のコードを読んで、どのような出力結果になるか、まずは、頭の中で予想してみましょう。予想が正しいかはスクリプトを動かして確認してみましょう。

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

```
print()

for i in range(5):
    print (str(i) + " I'm alive!")
print()

basket=["apple","peach","pear","cherry"]
for fruit in basket:
    print (fruit)
print()

my_color={"red":1,"blue":2,"green":3}
for color in my_color:
    print (color + " %d" % my_color[color]);
print()

name = "Brett"
if name == "Brett":
    print ("Brett who?")
else:
    print ("Nice name!")
```

実際にやってみよう

1. ディレクトリ \DevNetCode\<your-name>\devnet-express-code-samples\module04\04-lab04-learning-python-json に移動します。
2. Python スクリプト **loops.py** を実行します。次にファイルを開いてソース コードを確認します。確認したらファイルを閉じます。
3. Python スクリプト **loops_ex.py** を実行します。次にファイルを開いてソース コードを確認します。確認したらファイルを閉じます。
4. ファイル **json_parse-1.py** を開きます。ループを使用して Python コードを記述し、各 JSON オブジェクト内の**すべての**値を出力するように変更します。コードを実行します。

次のステップ: Python の関数を使用して全体をまとめましょう。

ステップ 6: 自分で関数を記述する

関数とは、明示的に呼び出された場合にのみ実行されるコードのまとまりです。たとえば print() は Python で記述される関数で、これまでにも何度も呼び出されています。関数は、コードをモジュール化するために記述されます。モジュール化することでコードが 1 ヵ所に保存され、再利用できるため、読みやすくデバッグも容易になるのです。本来、同じことを行うコードを何度も記述することは避けたいものです。これを関数にまとめることで、いつでも呼び出して実行できるようになります。

次に関数の構造を確認し、簡単な例を 1 つ見てみましょう。Python では、関数は次の方法で定義します。キーワード **def** は、関数が定義されていることを示し、その関数の名前と関数に渡されるオプションの引数が続きます。

```
def <function name>():
    code

def <function name>(arguments):
    code
```

関数の簡単な例を見てみましょう。1 つ目の関数 **my_function** は、固定の文字列を出力します。2 つ目の関数 **brett** は引数 **val** を受け取って、それを **range 関数** に渡し、ループで使用します。

```
def my_function():
        print("Hey I'm a function!")


def brett(val):
    for i in range(val):
        print("I'm a function with args!")
```

次に、これらの簡単な関数がスクリプト内でどのように呼び出されるかを見てみましょう。次のスクリプトが上から順番に実行されると、Python のインタプリタがどれを実行するか判断します。まず、Python のインタプリタは、1 行目にある print 関数の呼び出し部分を実行します。次に 2 つの関数が定義されているのを確認し、それが記録されますが、この時点で、この関数は定義されているだけで、明示的に呼び出されていないため、実行はされません。スクリプトが進むと、**my_function 関数**への呼び出し部分が確認され、それが実行されます。最後に **brett 関数**への呼び出しとその引数 5 が確認され、それが実行されます。

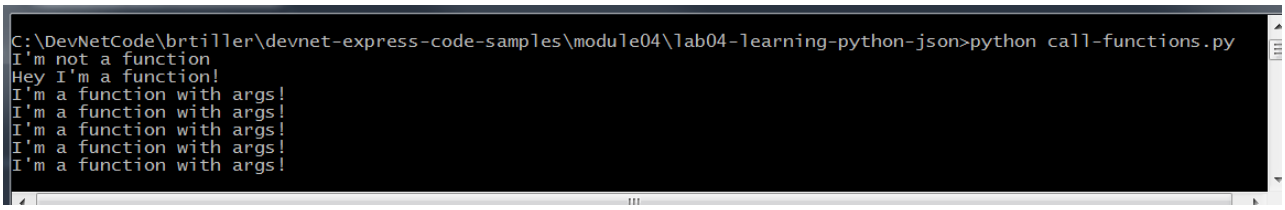
```
print ("I'm not a function")

def my_function():
        print("Hey I'm a function!")

def brett(val):
    for i in range(val):
        print("I'm a function with args!")

my_function()
brett(5)
```

その結果、出力は次のようになります。



最後のまとめ

このモジュールのこれまでのラボでは、Spark API を使用するスクリプトを実行しましたが、その時点では Python の記述方法については学習していませんでした。ここまでで、変数、演算子、スコープ、JSON の解析、ループ、関数についての基本事項を学習済みです。それぞれの関数と、その呼び出し方法を再確認してください。また、作成されたディクショナリと変数への割り当てを確認してください。

spark-room.py

このサンプル コードでは、Spark **rooms** REST API を使用して、アクセストークンに対応したユーザが属する Spark ルーム(スペース)のリストを取得します。

```
# requests ライブラリのインポート
import requests
```

```
# json ライブラリのインポート
import json

accessToken = "" # 自分のアクセストークンを""内に設定しましょう


def setHeaders():
    accessToken_hdr = 'Bearer ' + accessToken
    spark_header = {'Authorization': accessToken_hdr, 'Content-Type':
'application/json; charset=utf-8'}
    return spark_header


def getRooms(theHeader):
    uri = 'https://api.ciscospark.com/v1/rooms'
    resp = requests.get(uri, headers=theHeader)
    print("SparkAPI: ")
    return resp.json()


header=setHeaders()
value=getRooms(header)
#format output to make easier to read then display it.
print (json.dumps(value, indent=4, separators=(',', ': ')))
```

このサンプル コードは、次のように実行します。

1. ディレクトリ \DevNetCode\<your-name>\devnet-express-code-samples\module04\04-lab04-learning-python-json に移動します。
2. **spark-room.py** ファイルを開きます。たとえば、Windows では「**notepad spark-room.py**」と入力します。
3. 変数「accessToken」に Spark のアクセス トークンを割り当てます。すでに説明したように、アクセストークンを取得するには、URL https://developer.ciscospark.com/[英語] にアクセスしてログインし、Web ページの右上にあるメンバー アイコンをクリックします。
   ※ アクセストークンは誰にも公開しないようにしてください。
4. ファイルを保存します。エンコーディング タイプがオプションの場合は、**UTF-8** を選択します。
5. Python コマンドとファイル名 **spark-room.py** を入力し、エンター キーを押します。
6. プログラムが実行されるか、またはエラー メッセージが表示されます。

自身の属するルーム(スペース)が結果として表示されます。出力の例を以下に示します。簡略化のために、一部のレコードは割愛されています。



実際にやってみよう

1. ディレクトリ \DevNetCode\<your-name>\devnet-express-code-samples\module04\lab04-learning-python-json に移動します。
2. Python スクリプト **call-functions.py** を実行します。次に、ファイルを開いてソース コードを確認します。確認したらファイルを閉じます。
3. Python スクリプト **spark-room.py** を実行します。1 つ以上のルーム(スペース)に属する場合は、Python の 関数を定義して、その関数の中で、レスポンスの JSON を解析して各ルーム(スペース)の名前を表示するためのループを実装してみましょう。その後、コードを実行します。
4. Python スクリプト **spark-room-out.py** を開きます。Python の関数を定義して、その関数の中で、**rooms** 変数に割り当てられている各ルーム(スペース)のすべてのデータを解析して表示するループを実装してみましょう。その後、コードを実行します。

※ [teamId] フィールドと [id] フィールドの値は、プライバシ上の理由からダミーのものになっています。

おめでとうございます。Python によるコーディングと JSON の解析を完了しました。