Python チュートリアル学習ノート

これまでの学習を通じて、私たちは Python について少し理解を深めてきました。現在、公式ドキュメントに基づいて、Python の知識をさらに補足していきます。

コードフローの制御

タイプ

```
print(type(1))
```

このコードは、Python で整数 `1` のデータ型を出力します。実行すると、`<class 'int'>` と表示されます。これは、`1` が整数型（`int`）であることを示しています。

```
<class 'int'>
```

```
print(type('a'))
```

```
<class 'str'>
```

`type` 関数は、オブジェクトの型を表示するのに非常に便利です。

range

`range` は、Python で連続した数値のシーケンスを生成するために使用される関数です。主にループ処理で使われ、指定した範囲内の数値を順番に取り出すことができます。

基本的な使い方

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

このコードは、0 から 4 までの数値を順番に出力します。

開始値と終了値を指定する

```
for i in range(2, 6):
    print(i)
```

このコードは、2 から 5 までの数値を順番に出力します。

ステップを指定する

```
for i in range(1, 10, 2):
    print(i)
```

このコードは、1 から 9 までの数値を 2 つずつ飛ばして出力します（1, 3, 5, 7, 9）。

逆順に範囲を指定する

```
for i in range(10, 0, -1):
    print(i)
```

このコードは、10 から 1 までの数値を逆順に出力します。

range はメモリ効率が良いため、大きな範囲の数値を扱う場合にも適しています。

range 関数は非常に便利です。

```
for i in range(5):
  print(i, end = ' ')
```

このコードは、0 から 4 までの数字をスペース区切りで出力します。end = ' ' を指定することで、改行ではなくスペースが出力の区切り文字として使用されます。

```
0 1 2 3 4
```

```
for i in range(2, 6, 2):
  print(i, end = ' ')
```

このコードは、2 から始まり 6 未満の範囲で、2 ずつ増加する数値を出力します。end = ' ' は、出力の末尾にスペースを追加して、数値が同じ行に表示されるようにします。実行結果は次のようになります：

```
2 4
```

```
2 4
```

このコードブロックは、2 つの数字「2」と「4」を表示するだけのシンプルなシェルスクリプトの出力例です。特に翻訳の必要はありませんが、もし文脈上で何か説明が必要であれば、以下のように補足できます。

この出力は、シェルスクリプトが実行された結果として、2 つの数字「2」と「4」が表示されていることを示しています。

range 関数の定義を見てみましょう。

```
class range(Sequence[int]):
    start: int
    stop: int
    step: int
```

このコードは Python の range クラスを表しています。range は整数のシーケンスを生成するためのクラスで、start、stop、step の 3 つの属性を持っています。それぞれの属性は整数型（int）で、シーケンスの開始値、終了値、ステップ幅を指定します。

Visible はクラスです。

```
print(range(5))
```

このコードは、Python で range(5) を出力するものです。range(5) は 0 から 4 までの連続した整数を生成するイテレータを返しますが、print 関数で直接出力すると、range(0, 5) という形式で表示されます。これは、range オブジェクトそのものを表示しているためです。実際に数値のリストとして表示したい場合は、list(range(5)) のようにリストに変換する必要があります。

```
range(0, 5)
```

以下のように書くべきではありません：

```
[0,1,2,3,4]
```

続けます。

```
print(list(range(5)))
```

```
[0, 1, 2, 3, 4]
```

なぜでしょうか。list の定義を見てみましょう。

```
class list(MutableSequence[_T], Generic[_T]):
```

上記のコードは、Python の型ヒントを使用して、`list` クラスが `MutableSequence` と `Generic` を継承していることを示しています。`_T` はジェネリック型パラメータで、リストが任意の型の要素を保持できることを表しています。このコードは、リストがミュータブル（変更可能）なシーケンスであり、ジェネリック型であることを定義しています。

`list` の定義は `list(MutableSequence[_T], Generic[_T]):` です。一方で、`range` の定義は `class range(Sequence[int])` です。`list` は `MutableSequence` を継承しており、`range` は `Sequence` を継承しています。

続いて下を見ていくと、このようになっています。

```
Sequence = _alias(collections.abc.Sequence, 1)
MutableSequence = _alias(collections.abc.MutableSequence, 1)
```

このコードは、Python の `collections.abc` モジュールから `Sequence` と `MutableSequence` という抽象基底クラスをエイリアスとして定義しています。`_alias` 関数を使用して、これらのクラスを 1 つの型パラメータを持つジェネリック型として再定義しています。このコードは変更せずにそのまま使用されます。

ここでは、それらの関係がよくわかりません。しかし、なぜ `list(range(5))` と書けるのか、おおよそ理解できたと思います。

関数の引数

関数に関する補足知識を見ていきましょう。

```
def fn(a = 3):
  print(a)
```

このコードは、デフォルト引数を持つ関数 `fn` を定義しています。デフォルト引数 `a` の値は `3` で、関数が呼び出されたときに `a` の値が指定されない場合、`a` は自動的に `3` になります。関数内では、`a` の値を出力します。

fn()

```shell
3
```

（注：このコードブロック内の「3」は数値であり、翻訳の必要はありません。そのまま「3」と表示されます。）

これはパラメータにデフォルト値を与えるものです。

```
def fn(end: int, start = 1):
  i = start
  s = 0
  while i < end:
    s += i
    i += 1
  return s
```

この Python 関数 fn は、指定された範囲の整数の合計を計算します。end パラメータは範囲の終了値を指定し、start パラメータは範囲の開始値を指定します（デフォルトは 1）。関数は、start から end の直前までの整数を合計し、その結果を返します。

```
print(fn(10))
```

```
45
```

end は必須のパラメータです。必須のパラメータは最初に書くことに注意してください。

```
def fn(start = 1, end: int):
```

このコードスニペットは、Python の関数定義の一部です。fn という名前の関数を定義しており、2 つの引数を持っています。

- start はデフォルト値が 1 で、指定されない場合は 1 が使用されます。
- end は整数型（int）の引数で、デフォルト値は指定されていません。

ただし、このコードは不完全で、関数の本体が定義されていません。完全な関数定義にするには、関数の本体を追加する必要があります。

```
    def fn(start = 1, end: int):
                                ^
SyntaxError: デフォルト引数の後に非デフォルト引数が続いています
```

end が non-default argument（非デフォルト引数）であり、start が default argument（デフォルト引数）であることに注意してください。これは、非デフォルト引数がデフォルト引数の後に続いていることを意味します。つまり、非デフォルト引数はすべてのデフォルト引数の前に置く必要があります。start はデフォルト引数であり、もし値を渡さない場合、デフォルトで値が設定されていることを意味します。

```
def fn(a, /, b):
  print(a + b)
```

このPythonコードは、関数`fn`を定義しています。この関数は2つの引数`a`と`b`を受け取り、それらを足し合わせた結果を出力します。引数`a`の後にある`/`は、`a`が位置専用引数（positional-only argument）であることを示しています。つまり、`a`はキーワード引数として指定できません。

```
fn(1, 3)
```

ここで/を使ってパラメータの型を区切ります。パラメータの渡し方には2つの形式があります。1つは位置に基づいて渡す方法、もう1つはキーワードを指定して渡す方法です。

```
def fn(a, /, b):
  print(a + b)
```

このコードは、Pythonの関数定義を示しています。関数`fn`は2つの引数`a`と`b`を受け取ります。ここで、`/`は、`a`が位置専用引数（positional-only argument）であることを示しています。つまり、`a`はキーワード引数として指定できず、必ず位置引数として渡す必要があります。`b`は通常の引数で、位置引数またはキーワード引数として渡すことができます。関数内では、`a`と`b`の和を出力します。

```
fn(a=1, 3)
```

このコードは、Rustの関数呼び出しの構文に従っていません。Rustでは、関数の引数はカンマで区切られ、キーワード引数（`a=1`のような形式）はサポートされていません。正しい形式は以下のようになります：

```
fn(1, 3)
```

もし`a`が関数のパラメータ名である場合、関数定義時にパラメータ名を指定する必要がありますが、呼び出し時にはその名前は使用しません。例えば：

```
fn example(a: i32, b: i32) {
    // 関数の本体
}

example(1, 3);
```

このように、関数を定義し、呼び出す際にはパラメータ名を指定せずに値を渡します。

```
    fn(a=1, 3)
             ^
SyntaxError: キーワード引数の後に位置引数が続いています
```

このように書くのはうまくいきません。`a=1` は、これがキーワードで引数を渡すことを意味します。これはキーワード引数として扱われます。一方、`b` は位置引数です。

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                  |
        |        位置またはキーワード      |
        |                                - キーワードのみ
         -- 位置のみ
```

この Python の関数定義では、引数の渡し方に制約が設けられています。`/`の前にある引数（`pos1` と `pos2`）は位置引数としてのみ渡すことができ、キーワード引数として渡すことはできません。`/` と `*` の間にある引数（`pos_or_kwd`）は、位置引数としてもキーワード引数としても渡すことができます。`*` の後にある引数（`kwd1` と `kwd2`）はキーワード引数としてのみ渡すことができ、位置引数として渡すことはできません。

ここで関数を定義する際に、`/` と `*` を使用することで、各パラメータの受け渡しタイプが暗黙的に指定されています。したがって、規則に従ってパラメータを渡す必要があります。

```
def fn(a, /, b):
  print(a + b)
```

このコードは、Python の関数定義を示しています。関数 `fn` は 2 つの引数 `a` と `b` を受け取ります。`/` は、Python 3.8 で導入された構文で、`/` の前にある引数（この場合は `a`）は位置専用引数（positional-only argument）であることを示します。つまり、`a` はキーワード引数として指定できません。`b` は位置引数またはキーワード引数として指定できます。

この関数は、`a` と `b` を足し合わせた結果を出力します。

```
fn(1, b=3)
```

上記のようにするとエラーは発生しませんでした。

```
def fn(a, /, b, *, c):
  print(a + b + c)
```

この Python 関数は、以下のように定義されています：

- a は位置専用引数（positional-only argument）です。これは、関数を呼び出す際にキーワード引数として指定できないことを意味します。
- b は位置引数またはキーワード引数として指定できます。
- c はキーワード専用引数（keyword-only argument）です。これは、関数を呼び出す際にキーワード引数としてのみ指定できることを意味します。

この関数は、a、b、c の 3 つの引数を足し合わせた結果を出力します。

fn(1, 3, 4)

```shell
    fn(1, 3, 4)
TypeError: fn() は2つの位置引数を取りますが、3つが与えられました
```

fn は 2 つの位置引数しか受け取れませんが、3 つが与えられました。

```
def fn(a, /, b, *, c):
  print(a + b + c)
```

この Python の関数定義では、/と * が引数の区切りとして使用されています。それぞれの意味は以下の通りです：

- /：/の前にある引数（この場合は a）は、位置専用引数（positional-only arguments）です。つまり、キーワード引数として指定することはできません。
- *：* の後にある引数（この場合は c）は、キーワード専用引数（keyword-only arguments）です。つまり、位置引数として指定することはできません。

したがって、この関数は以下のように呼び出すことができます：

```
fn(1, 2, c=3)  # 正しい呼び出し方
```

しかし、以下の呼び出し方はエラーになります：

```
fn(a=1, b=2, c=3)  # エラー: a は位置専用引数なのでキーワード引数として指定できない
fn(1, 2, 3)        # エラー: c はキーワード専用引数なので位置引数として指定できない
```

このように、/と * を使うことで、関数の引数の渡し方を制限することができます。

```
fn(a = 1, b=3, c=4)
```

```
    fn(a = 1, b=3, c=4)
TypeError: fn() にキーワード引数として渡された位置専用引数があります: 'a'
```

fn には、以前は位置によってのみ渡すことができたパラメータが、現在ではキーワードを使って渡すようになっています。

マップ形式のパラメーター

```
def fn(**kwds):
  print(kwds)

fn(**{'a': 1})
```

上記のコードは、Python の関数呼び出しにおいて、辞書をキーワード引数として展開する方法を示しています。** 演算子を使用して、辞書のキーと値を関数のキーワード引数として渡しています。この場合、fn 関数は a=1 というキーワード引数を受け取ります。

```
{'a': 1}
```

(注：コードブロック内の内容は翻訳しないでください。)

```
def fn(**kwds):
  print(kwds['a'])
```

この Python コードは、キーワード引数を辞書として受け取る関数 fn を定義しています。**kwds は、関数に渡されたすべてのキーワード引数を辞書としてまとめます。この例では、kwds['a'] を出力しています。つまり、関数が呼び出されたときに a というキーワード引数が渡されると、その値が出力されます。

```
d = {'a': 1}
fn(**d)
```

上記のコードは、Python の辞書 d を関数 fn にキーワード引数として展開して渡す方法を示しています。**d は辞書のキーと値をキーワード引数として展開します。つまり、fn(a=1) と同等の呼び出しが行われます。

```
1
```

（このコードブロックは単に数字の「1」を表示しているだけなので、翻訳の必要はありません。）

** はパラメータを展開するためのものです。

```
def fn(a, **kwds):
  print(kwds['a'])
```

この Python コードは、関数 `fn` を定義しています。この関数は、1 つの必須引数 `a` と、任意の数のキーワード引数 `**kwds` を受け取ります。関数内では、キーワード引数 `kwds` の中からキー`'a'`に対応する値を出力します。

ただし、このコードには潜在的な問題があります。もし `kwds` にキー`'a'`が含まれていない場合、`KeyError` が発生します。そのため、実際に使用する際には、`kwds` に`'a'`が含まれているかどうかを確認するか、デフォルト値を指定するなどの対策が必要です。

```
d = {'a': 1}
fn(1, **d)
```

このコードは、Python の辞書 `d` を関数 `fn` にキーワード引数として渡す方法を示しています。`**d` は辞書 `d` を展開し、そのキーと値をキーワード引数として関数に渡します。したがって、`fn(1, **d)` は `fn(1, a=1)` と等価です。

```
TypeError: fn() で引数 'a' に複数の値が指定されました
```

`fn(1, **d)` のように関数を呼び出すと、展開されて `fn(a=1, a=1)` となります。そのため、エラーが発生します。

```
def fn(**kwds):
  print(kwds['a'])
```

このコードは、Python の関数 `fn` を定義しています。この関数は、キーワード引数を辞書として受け取り、その中からキー `'a'` に対応する値を出力します。

```
d = {'a': 1}
fn(d)
```

```
TypeError: fn() は 0 個の位置引数を取りますが、1 個が与えられました
```

もし `fn(d)` のように関数を呼び出すと、それはキーワード引数として展開されるのではなく、位置引数として扱われます。

```
def fn(a, / , **kwds):
  print(kwds['a'])
```

このコードは、Python の関数定義を示しています。関数 `fn` は、第一引数 `a` を位置専用引数として受け取り、それ以降の引数をキーワード引数として `kwds` 辞書に格納します。関数内では、`kwds` 辞書からキー `'a'` に対応する値を出力します。

ただし、このコードには潜在的な問題があります。関数が呼び出されるときに、`a` がキーワード引数として渡されない場合、`KeyError` が発生します。例えば、以下のように呼び出した場合です：

```
fn(1, b=2)  # KeyError: 'a'
```

このエラーを防ぐためには、`kwds` 辞書に `'a'` が含まれているかどうかを確認する必要があります。

```
d = {'a': 1}
fn(1, **d)
```

このコードは、Python の辞書 `d` を関数 `fn` にキーワード引数として渡す方法を示しています。`**d` は辞書 `d` を展開し、そのキーと値をキーワード引数として関数に渡します。この場合、`fn(1, a=1)` と同等の呼び出しが行われます。

こうすればうまくいきます。位置引数とマップ形式の引数が同じ名前でも問題ないことがわかります。

```
def fn(a, / , a):
  print(a)
```

このコードは、Python の関数定義を示していますが、いくつかの問題があります。

1. **引数の重複**: 関数 `fn` の引数リストに `a` が 2 回指定されています。Python では、関数の引数名は一意でなければなりません。同じ名前の引数を複数回指定することはできません。
2. **位置専用引数**: 引数リストの最初の `a` の後に `/` が置かれています。これは、`/` より前の引数が位置専用引数（positional-only arguments）であることを示します。位置専用引数は、キーワード引数として指定することができません。しかし、この場合、`a` が重複しているため、この構文は無効です。

このコードを修正するには、引数名を一意にする必要があります。例えば、以下のように修正できます：

```
def fn(a, /, b):
    print(a, b)
```

この修正により、`a` は位置専用引数、`b` は位置またはキーワード引数として指定できるようになります。

```
d = {'a': 1}
fn(1, **d)
```

上記のコードは、Python の辞書 `d` を定義し、その辞書をキーワード引数として関数 `fn` に渡しています。`**d` は辞書のキーと値をキーワード引数として展開するための構文です。この場合、`fn(1, a=1)` と同等の呼び出しが行われます。

```
SyntaxError: 関数定義で引数 'a' が重複しています
```

このようにするとエラーが発生します。これらの状況の微妙な関係に注意してください。

```
def fn(a, / , **kwds):
  print(kwds['a'])
```

このコードは、Python の関数定義を示しています。ここで、`/`は、それ以前の引数が位置専用引数であることを示しています。つまり、`a` はキーワード引数として渡すことができません。`**kwds` は、キーワード引数を辞書として受け取ります。この関数は、`kwds` 辞書内のキー`'a'`に対応する値を出力します。

`fn(1, **[1,2])` は、Python の関数呼び出しにおいて、リスト `[1, 2]` をアンパックして引数として渡す構文です。具体的には、`**` 演算子を使用して、リストの要素をキーワード引数として展開します。

ただし、このコードは正しく動作しません。なぜなら、`**` 演算子は辞書に対して使用されるべきであり、リストに対しては使用できないからです。正しい使い方は、辞書をアンパックする場合です。例えば、以下のように書くことができます：

```
def fn(a, b, c):
    print(a, b, c)

fn(1, **{'b': 2, 'c': 3})
```

この場合、`fn(1, **{'b': 2, 'c': 3})` は `fn(1, b=2, c=3)` と等価になります。

もしリストの要素を位置引数として展開したい場合は、`*` 演算子を使用します：

```
def fn(a, b, c):
    print(a, b, c)

fn(1, *[2, 3])
```

この場合、`fn(1, *[2, 3])` は `fn(1, 2, 3)` と等価になります。

```
TypeError: __main__.fn() の ** の後の引数はマッピングでなければなりませんが、リストが指定されました
```

`**` の後にはマッピングが続く必要があります。

イテラブル型のパラメータ

```
def fn(*kwds):
  print(kwds)
```

`fn(*[1, 2])` は、Python における関数呼び出しの一形態です。このコードは、リスト `[1, 2]` をアンパックして、その要素を個別の引数として関数 `fn` に渡します。具体的には、`fn(1, 2)` と同じ意味になります。

以下にコードブロックで示します：

```
fn(*[1, 2])
```

このコードは、リスト `[1, 2]` の要素を展開して、関数 `fn` に引数として渡します。

```
(1, 2)
```

注: このコードブロックはそのまま表示されます。特に翻訳する必要はありません。

```
def fn(*kwds):
  print(kwds)

fn(*1)
```

```shell
TypeError: __main__.fn() の * の後の引数はイテラブルでなければなりませんが、int が指定されました
```

* は iterable に続く必要があります。

```
def fn(a, *kwds):
  print(type(kwds))
```

このコードは、関数 fn を定義しています。この関数は、少なくとも 1 つの引数 a を受け取り、それに続いて任意の数のキーワード引数 *kwds を受け取ります。*kwds は、関数に渡された追加の引数をタプルとして保持します。print(type(kwds)) は、kwds のデータ型を出力します。この場合、kwds はタプル型 (<class 'tuple'>) として表示されます。

fn(1, *[1])

```shell
<class 'tuple'>
```

型を出力してみましょう。これが、上記で [1,2] ではなく (1,2) が出力される理由です。

```
def fn(*kwds):
  print(kwds)
```

fn(1, *[1])

```shell
(1, 1)
```

ここで fn(1, *[1]) を呼び出すと、引数が展開されて fn(1,1) になります。その後、fn(*kwds) が解析されるとき、kwds は 1,1 をタプル (1,1) に変換します。

```
def concat(*args, sep='/'):
  return sep.join(args)
```

この Python 関数は、可変長の引数 *args を受け取り、それらを指定された区切り文字 sep（デフォルトは '/'）で結合して返します。例えば、concat('a', 'b', 'c') は 'a/b/c' を返します。

```
print(concat('a','b','c', sep=','))
```

```
a,b,c
```

(注：コードブロック内の内容はそのまま保持されます。翻訳の必要はありません。)

Lambda 式

lambda は、関数を変数として保存する方法です。「コンピュータサイエンスの謎解き」という記事で述べたことを覚えていますか？

```
def incrementor(n):
  return lambda x: x + n
```

この Python コードは、incrementor という関数を定義しています。この関数は、引数として数値 n を受け取り、lambda 式を使って新しい関数を返します。返される関数は、引数 x を受け取り、x に n を加えた結果を返します。つまり、incrementor は特定の数 n だけ増加させる関数を生成するファクトリ関数として機能します。

```
f = incrementor(2)
print(f(3))
```

```
5
```

(注：このコードブロック内の「5」は数値であり、翻訳の必要はありません。そのまま「5」と表示されます。)

もう一つの例を見てみましょう。

```
pairs = [(1, 4), (2, 1), (0, 3)]

pairs.sort(key = lambda pair: pair[1])

print(pairs)
```

```
[(2, 1), (0, 3), (1, 4)]
```

(注：コードブロック内の内容はそのまま保持されます。これは数値のリストであり、翻訳の対象ではありません。)

```
pairs = [(1, 4), (2, 1), (0, 3)]
```

このコードは、Python でタプルのリストを作成しています。各タプルは 2 つの整数を含んでいます。具体的には、(1, 4)、(2, 1)、(0, 3) という 3 つのタプルがリストに含まれています。

```
pairs.sort(key = lambda pair: pair[0])
```

```
print(pairs)
```

```
[(0, 3), (1, 4), (2, 1)]
```

pair[0] の場合、最初の数でソートします。pair[1] の場合、2 番目の数でソートします。

ドキュメンテーションコメント

```
def add():
  """ 何かを追加する
  """
  pass

print(add.__doc__)
```

何かを追加

関数シグネチャ

```
def add(a:int, b:int) -> int:
  print(add.__annotations__)
  return a+b
```

この Python コードは、add という関数を定義しています。この関数は 2 つの整数 a と b を受け取り、それらを足し合わせた結果を返します。また、関数のアノテーション（型ヒント）を表示するために、add.__annotations__ を出力しています。

- a:int と b:int は、関数の引数 a と b が整数型であることを示しています。
- -> int は、関数の戻り値が整数型であることを示しています。
- add.__annotations__ は、関数のアノテーション（型ヒント）を辞書形式で返します。この場合、{'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>} のような出力が得られます。

```
add(1, 2)
```

```
{'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

このコードブロックは、Python の型アノテーションを示しています。a と b は整数型（int）であり、返り値も整数型（int）であることを表しています。

データ構造

リスト

```
a = [1,2,3,4]
```

このコードは、Python でリスト a を定義しています。リスト a には、整数 1, 2, 3, 4 が順番に格納されています。

```
a.append(5)
print(a)   # [1, 2, 3, 4, 5]

a[len(a):] = [6]
print(a)   # [1, 2, 3, 4, 5, 6]
```

このコードは、リスト a の末尾に新しい要素 6 を追加します。a[len(a):] はリストの末尾を指し、そこに [6] を代入することで、リストの末尾に 6 が追加されます。結果として、a は [1, 2, 3, 4, 5, 6] となります。

```
a[3:] = [6]
print(a)   # [1, 2, 3, 6]

a.insert(0, -1)
print(a)   # [-1, 1, 2, 3, 6]

a.remove(1)
print(a)   # [-1, 2, 3, 6]

a.pop()
print(a)  # [-1, 2, 3]

a.clear()
print(a)  # []

a[:] = [1, 2]
print(a.count(1)) # 1
```

このコードは、リスト a のすべての要素を [1, 2] に置き換え、その後、リスト内の 1 の出現回数をカウントして出力します。結果として、1 はリスト内に 1 回だけ出現するため、出力は 1 となります。

```
a.reverse()
print(a)   # [2, 1]

b = a.copy()
a[0] = 10
print(b)   # [2, 1]
print(a)   # [10, 1]

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

リストの構築

```
print(3 ** 2)   # 9
print(3 ** 3)   # 27
```

まず、一つの演算子を学びましょう。それは `**` です。これはべき乗を意味します。

```
sq = []
for x in range(10):
  sq.append(x ** 2)

print(sq)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

次に、`map` を使ってみましょう。

```
a = map(lambda x:x, range(10))
print(a)
# <map object at 0x103bb0550>
print(list(a))
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

sq = map(lambda x: x ** 2, range(10))
print(list(sq))
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
sq = [x ** 2 for x in range(10)]
print(sq)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

for は非常に柔軟であることがわかります。

```
a = [i for i in range(5)]
print(a)
# [0, 1, 2, 3, 4]

a = [i+j for i in range(3) for j in range(3)]
print(a)
# [0, 1, 2, 1, 2, 3, 2, 3, 4]

a = [i for i in range(5) if i % 2 == 0]
print(a)
# [0, 2, 4]

a = [(i,i) for i in range(3)]
print(a)
# [(0, 0), (1, 1), (2, 2)]
```

ネストされたリストの構築

```
matrix = [[(i+j*4) for i in range(4)] for j in range(3)]
print(matrix)
# [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

このコードは、3 行 4 列の行列を作成し、その内容を表示します。各行は 0 から始まる連続した数値で構成され、次の行は前の行の最後の数値に 1 を加えた値から始まります。結果として、以下のような行列が出力されます。

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

t = []
for j in range(3):
  t.append([(i+j*4) for i in range(4)])
print(t)
# [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

以下の 2 つのコードの方法に注意してください。つまり：

```
[[(i+j*4) for i in range(4)] for j in range(3)]
```

この Python のリスト内包表記は、次のような 2 次元リストを生成します。各要素は `i + j * 4` で計算され、`i` は 0 から 3 までの範囲、`j` は 0 から 2 までの範囲で繰り返されます。

結果として得られるリストは以下のようになります：

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

このコードは、3 行 4 列の行列を作成し、各要素を行と列のインデックスに基づいて計算しています。

これは以下と同等です：

```
for j in range(3):
   [(i+j*4) for i in range(4)]
```

このコードは、Python のリスト内包表記を使用して、特定のパターンで数値を生成しています。具体的には、`j` が 0 から 2 までの範囲でループし、各 `j` に対して `i` が 0 から 3 までの範囲でループします。各 `i` に対して、`i + j * 4` の値を計算し、その結果をリストに格納します。

このコードを実行すると、以下のようなリストが生成されます：

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

各内側のリストは、`j` の値に応じて 4 ずつ増加する数値のシーケンスを表しています。

つまり、以下のように相当します：

```
for j in range(3):
  for i in range(4):
      (i+j*4)
```

このコードは、Python でネストされたループを使用して、特定の計算を行っています。具体的には、外側のループが `j` を 0 から 2 まで、内側のループが `i` を 0 から 3 まで繰り返します。各イテレーションで、`i + j * 4` の値が計算されますが、この値はどこにも保存されず、単に計算されるだけです。したがって、このコードは何も出力しません。もし結果を表示したい場合は、`print(i + j * 4)` のように変更する必要があります。

したがって、これは行列の転置を行うのに便利です。

```
matrix = [[(i+j*4) for i in range(4)] for j in range(3)]
print(matrix)
#  [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

このコードは、3 行 4 列の行列を作成し、その内容を表示します。行列の各要素は、行インデックス `j` と列インデックス `i` に基づいて計算されます。具体的には、`i + j * 4` という式で各要素が生成されます。結果として、以下のような行列が出力されます：

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

この行列は、3 つの行と 4 つの列を持ち、各要素は 0 から 11 までの連続した整数で構成されています。

```
mt = [[row[j] for row in matrix] for j in range(4)]
print(mt)
# [[0, 4, 8], [1, 5, 9], [2, 6, 10], [3, 7, 11]]

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

del

`del` は、Python の組み込み関数で、指定したオブジェクトを削除するために使用されます。主にリストや辞書などのミュータブルなデータ構造から要素を削除する際に利用されます。以下にいくつかの使用例を示します。

リストからの要素の削除

```
my_list = [1, 2, 3, 4, 5]
del my_list[2]  # インデックス 2 の要素を削除
print(my_list)  # 出力: [1, 2, 4, 5]
```

辞書からのキーの削除

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
del my_dict['b']  # キー 'b' を削除
print(my_dict)  # 出力: {'a': 1, 'c': 3}
```

変数の削除

```
x = 10
del x  # 変数 x を削除
print(x)  # NameError: name 'x' is not defined
```

del を使用することで、不要なオブジェクトをメモリから解放し、リソースを効率的に管理することができます。ただし、削除したオブジェクトにアクセスしようとするとエラーが発生するため、注意が必要です。

```
a = [1, 2, 3, 4]
```

このコードは、Python でリスト a を定義しています。リスト a には、整数 1, 2, 3, 4 が順番に格納されています。

```
del a[1]
print(a)  # [1, 3, 4]

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

このコードは、リスト a の最初の 2 つの要素を削除し、残りの要素を表示します。結果として、リスト a には [4] だけが残ります。

```
del a
print(a) # NameError: name 'a' is not defined
```

上記のコードを日本語で説明すると、以下のようになります。

```
del a
print(a) # NameError: 名前 'a' は定義されていません
```

このコードでは、変数 a を del ステートメントで削除した後、その変数を参照しようとしています。しかし、a はすでに削除されているため、NameError が発生し、「名前 'a' は定義されていません」というエラーメッセージが表示されます。

辞書

```
ages = {'li': 19, 'wang': 28, 'he' : 7}
for name, age in ages.items():
    print(name, age)
```

このコードは、ages という辞書に含まれる各キーと値を順に取り出し、名前と年齢を出力します。具体的には、li、wang、he という名前とそれぞれの年齢が表示されます。

li 19

wang 28

he 7

```
for name in ages:
    print(name)

# li
# wang
# he

for name, age in ages:
    print(name)
```

ValueError: アンパックする値が多すぎます（期待される数は 2 です）

```
for i, name in enumerate(['li', 'wang', 'he']):
    print(i, name)

# 0 李
# 1 王
# 2 何
```

print(reversed([1, 2, 3])) # <list_reverseiterator object at 0x10701ffd0>

```
print(list(reversed([1, 2, 3])))
# [3, 2, 1]
```

あなたはプロの翻訳者です。Jekyll ブログ投稿用のマークダウンファイルを翻訳しています。以下のテキスト

モジュール

スクリプト方式でのモジュール呼び出し

```
import sys

def f(n):
    if n < 2:
        return n
    else:
        return f(n-1) + f(n-2)

if __name__ == "__main__":
    r = f(int(sys.argv[1]))
    print(r)
```

```
% python fib.py 3
2
```

このコードは、fib.py という Python スクリプトを実行し、引数として 3 を渡しています。実行結果として 2 が出力されています。これは、おそらくフィボナッチ数列の第 3 項を計算しているものと思われます。

```
% python -m fib 5
5
```

このコードは、Python の fib モジュールを実行し、引数として 5 を渡しています。実行結果として 5 が出力されています。

dir

dir コマンドは、指定されたディレクトリ内のファイルとサブディレクトリの一覧を表示するために使用されます。このコマンドは、Windows コマンドプロンプトでよく使用されます。

使用例

```
dir
```

このコマンドを実行すると、現在のディレクトリ内のすべてのファイルとサブディレクトリが表示されます。

オプション

- /p: 一画面ごとに表示します。
- /w: ワイド形式で表示します。
- /s: 指定されたディレクトリとそのすべてのサブディレクトリ内のファイルを表示します。

```
dir /p
```

このコマンドを実行すると、一画面ごとにファイルとディレクトリの一覧が表示されます。

dir コマンドは、ディレクトリの内容を確認するための基本的で便利なツールです。

```
import fib

print(dir(fib))
```

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__
```

このコードブロックは Python のモジュールやスクリプトで使用される特殊な属性や変数のリストを示しています。これらの属性は、Python インタプリタによって自動的に設定されるもので、翻訳の必要はありません。

```
import builtins
print(dir(builtins))
```

このコードは、Python の組み込み関数やオブジェクトを管理する builtins モジュールをインポートし、その中に含まれるすべての属性や関数のリストを表示します。dir() 関数は、指定されたオブジェクトの有効な属性のリストを返します。この場合、builtins モジュールのすべての属性が表示されます。

[‘ArithmeticError’, ‘AssertionError’, ‘AttributeError’, ‘BaseException’, ‘BlockingIOError’, ‘BrokenPipeError’, ‘BufferError’, ‘BytesWarning’, ‘ChildProcessError’, ‘ConnectionAbortedError’, ‘ConnectionError’, ‘ConnectionRefusedError’, ‘ConnectionResetError’, ‘DeprecationWarning’, ‘EOFError’,

‘Ellipsis’, ‘EnvironmentError’, ‘Exception’, ‘False’, ‘FileExistsError’, ‘FileNotFoundError’, ‘FloatingPointError’, ‘FutureWarning’, ‘GeneratorExit’, ‘IOError’, ‘ImportError’, ‘ImportWarning’, ‘IndentationError’, ‘IndexError’, ‘InterruptedError’, ‘IsADirectoryError’, ‘KeyError’, ‘KeyboardInterrupt’, ‘LookupError’, ‘MemoryError’, ‘ModuleNotFoundError’, ‘NameError’, ‘None’, ‘NotADirectoryError’, ‘NotImplemented’, ‘NotImplementedError’, ‘OSError’, ‘OverflowError’, ‘PendingDeprecationWarning’, ‘PermissionError’, ‘ProcessLookupError’, ‘RecursionError’, ‘ReferenceError’, ‘ResourceWarning’, ‘RuntimeError’, ‘RuntimeWarning’, ‘StopAsyncIteration’, ‘StopIteration’, ‘SyntaxError’, ‘SyntaxWarning’, ‘SystemError’, ‘SystemExit’, ‘TabError’, ‘TimeoutError’, ‘True’, ‘TypeError’, ‘UnboundLocalError’, ‘UnicodeDecodeError’, ‘UnicodeEncodeError’, ‘UnicodeError’, ‘UnicodeTranslateError’, ‘UnicodeWarning’, ‘UserWarning’, ‘ValueError’, ‘Warning’, ‘ZeroDivisionError’, ‘**build_class**’, ‘**debug**’, ‘**doc**’, ‘**import**’, ‘**loader**’, ‘**name**’, ‘**package**’, ‘**spec**’, ‘abs’, ‘all’, ‘any’, ‘ascii’, ‘bin’, ‘bool’, ‘breakpoint’, ‘bytearray’, ‘bytes’, ‘callable’, ‘chr’, ‘classmethod’, ‘compile’, ‘complex’, ‘copyright’, ‘credits’, ‘delattr’, ‘dict’, ‘dir’, ‘divmod’, ‘enumerate’, ‘eval’, ‘exec’, ‘exit’, ‘filter’, ‘float’, ‘format’, ‘frozenset’, ‘getattr’, ‘globals’, ‘hasattr’, ‘hash’, ‘help’, ‘hex’, ‘id’, ‘input’, ‘int’, ‘isinstance’, ‘issubclass’, ‘iter’, ‘len’, ‘license’, ‘list’, ‘locals’, ‘map’, ‘max’, ‘memoryview’, ‘min’, ‘next’, ‘object’, ‘oct’, ‘open’, ‘ord’, ‘pow’, ‘print’, ‘property’, ‘quit’, ‘range’, ‘repr’, ‘reversed’, ‘round’, ‘set’, ‘setattr’, ‘slice’, ‘sorted’, ‘staticmethod’, ‘str’, ‘sum’, ‘super’, ‘tuple’, ‘type’, ‘vars’, ‘zip’]

パッケージ

パッケージ、すなわち`packages`。

```shell
pk.py
fibp
   cal
       cal.py
   pt
        pt.py
```

（注：この部分はディレクトリ構造を示すコードブロックであり、翻訳の必要はありません。）

```python
# cal.py

def f(n):
    if n < 2:
        return n
    else:
```

```
        return f(n-1) + f(n-2)

def fl(n):
    return list(map(f, range(5)))
```

この Python コードは、フィボナッチ数列を計算する再帰関数 `f(n)` と、その関数を使って最初の 5 つのフィボナッチ数をリストとして返す関数 `fl(n)` を定義しています。コード自体は日本語に翻訳する必要はありませんが、説明を日本語で補足すると以下のようになります。

- `f(n)` は、与えられた整数 `n` に対してフィボナッチ数列の第 `n` 項を返します。`n` が 2 未満の場合は `n` をそのまま返し、それ以外の場合は `f(n-1)` と `f(n-2)` の和を返します。
- `fl(n)` は、`range(5)` で生成される 0 から 4 までの整数に対して `f` 関数を適用し、その結果をリストとして返します。つまり、最初の 5 つのフィボナッチ数 `[0, 1, 1, 2, 3]` を返します。

このコードは、再帰的なアプローチでフィボナッチ数列を計算する典型的な例です。

pt.py:

```
def p(l):
    print(l, end=' ')

def pln(l):
    print(l)
```

```
# pk.py

import fibp.cal.cal
import fibp.pt.pt

fibp.pt.pt.p(fibp.cal.cal.fl(10))
```

`pk.py`は次のように書くこともできます：

```
```python
from fibp.cal import cal
from fibp.pt import pt

pt.p(cal.fl(10))
```
```