Pythonの処理系はどのように実装され，どのように動いているのか？ 我々はその実態を調査すべくアマゾンへと飛んだ.

KMC春合宿2017 @utgwkk

私について

- @utgwkk
- KMC-ID: utgw
- 京都大学工学部情報学科
  計算機科学コース2回生
- デレステ: 581948763

講座の対象・予定

- C言語のような静的型付け・コンパイル言語しか触ったことがない人向け
- 動的型付け・インタプリタ言語の中身が気になる人向け
- Python の標準実装である CPython がどう実装されているのか見ていく
  - 他の実装もだいたいこんな感じ？
- あまり深いところまで行かずにざっと見ていく予定
  - 私の理解が追い付いていないので

今日話すこと

1. Python の概要
2. Python のオブジェクトの構造体による内部表現
3. Python の主要なデータ型の実装
4. Python の関数・モジュールの実装
5. Python のコードの実行・仮想機械について

今日話すこと

1. Python の概要
2. Python のオブジェクトの構造体による内部表現
3. Python の主要なデータ型の実装
4. Python の関数・モジュールの実装

どのように実装され

5. Python のコードの実行・仮想機械について

どのように動いているのか

Python の概要

Python とは

- Guido van Rossum が作ったプログラミング言語
- 動的型付けプログラミング言語
- インタプリタ言語
- 最新バージョンは 3.6 (2016/12/23)

Python の特徴

- 動的型付けプログラミング言語である
  - 型ヒントが最近導入された
- インタプリタ言語である
- インデントによってブロックを表現する (オフサイドルール)
- コードがシンプルで扱いやすく設計されている
- 機械学習・統計ライブラリが豊富
  - 日本だと機械学習目的で書いてる人が多そう [要出典]

Python のコードの例

- 変数の定義

```
x = 12345 # int(整数型)
st = “this is string” # str(文字列型)
bo = True # bool(真偽値)
obj = None # 何もない的な
```

Python のコードの例

- 変数の定義

```
alist = [3, 4, 6]
atuple = (3, 4, 6)
adict = {‘hoge’: 10, ‘fuga: 20}
aset = {1, 3, 5, 7}
```

Python のコードの例

- 関数の定義 (def 文)

```
def process(x, y=5):
    return x + y - 1
```

Python のコードの例

- 条件分岐 (if 文)

```
if num < 50:
    print(“less than 50.”)
elif num < 100:
    print(“less than 100.”)
else:
    print(“sugo-i!!!!!!!”)
```

Python のコードの例

- 繰り返し (for 文, while 文)

```
for i in range(10):
    print(i ** 2)

while i < 50:
    i *= 2
```

Python のコードの例

- 例外処理

```
try:
    do_something_abunai()
except NanikaError as e:
    print('例外が発生し，捕捉される場合')
else:
    print('例外が発生しなかった場合')
finally:
    print('いずれの場合にも実行される処理')
```

Python のコードの例

- コンテキストマネージャ
  - あるオブジェクトに対する初期化と後片付けをまとめて記述できる
    - Go の defer, C# の using 文と似ている
  - open() されたファイルは with 文を抜ける際に必ず close() される

```
with open(“hoge.txt”, “wt”) as f:
    f.write(data)
```

Python のコードの例

- コンテキストマネージャ
  - だいたいこのコードと等価

```
f = open(“hoge.txt”, “wt”)
try:
    f.write(data)
finally:
    f.close()
```

Python のコードの例

- ジェネレータ
  - なんらかの処理をした値を複数回返すというときに便利
  - 遅延評価イテレータを簡単に作れる

```
def gen():
    data = do_something()
    yield data
    yield do_something_else(data)
```

Python のコードの例

- 値のスワップ
  - これでいい
  - 一時変数は使わないの？
    - あとで見る

```
x, y = y, x
```

Python のコードの例

- 内包表記

```
[x ** 2 for x in range(100) if x % 2 == 0]
{x ** 2 for x in range(100) if x % 2 == 0}
{str(x): x for x in range(100) if x % 2 == 0}
(x ** 2 for x in range(100) if x % 2 == 0) # ジェネレータ
```

Python のオブジェクトの構造体による内部表現

CPython

- van Rossum が書いた Python の標準実装
- C言語で記述されている
- CPython を使ってライブラリを書くことができる
  - いわゆるC拡張ですね

Python のオブジェクトの実装

- オブジェクトを表す構造体
  - PyObject
  - PyVarObject
- クラスを表す構造体
  - PyTypeObject

オブジェクトを表す2つの構造体

Includes/object.h L106-115

```
typedef struct _object {
    _PyObject_HEAD_EXTRA
    Py_ssize_t ob_refcnt;
    struct _typeobject *ob_type;
} PyObject;
```

```
typedef struct {
    PyObject ob_base;
    Py_ssize_t ob_size;
} PyVarObject;
```

```
#define _PyObject_HEAD_EXTRA
```

```
PyVarObject extends PyObject
```

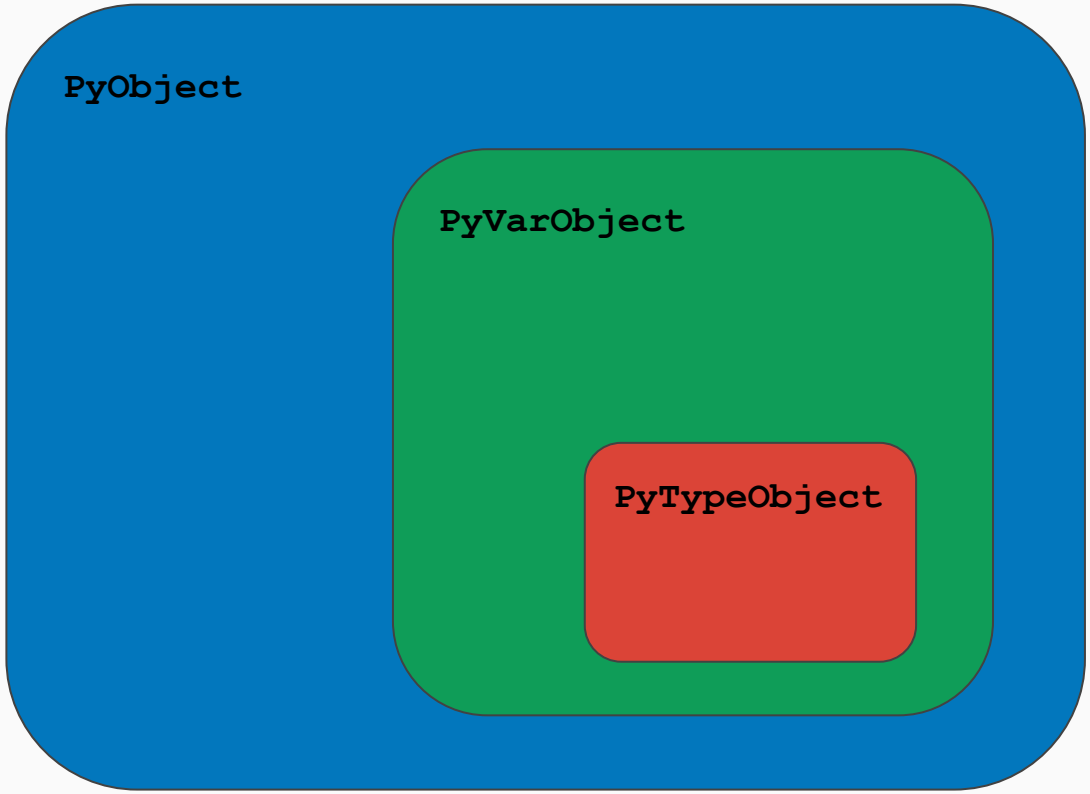
Py(Var)Object の持つ要素

- オブジェクトの参照カウンタ (ob_refcnt)
- PyTypeObject へのポインタ (*ob_type)
- (可変長オブジェクトの要素数 (ob_size))

PyTypeObject

- Doc/includes/typestruct.h
- Python のクラスを表す構造体
- メンバがいっぱいある
  - 演算子オーバーロードとか
  - メンバ関数，メンバ変数
  - ハッシュ関数

関係性



この他にもいろいろな構造体があります

- おいおい出てくるのでその都度やっていきます
  - データ構造を表す構造体
  - メンバ関数を表す構造体
  - モジュールを表す構造体
  - etc.

Python の主要なデータ型の実装

Python のデータ型

- 組み込みデータ型
  - int (整数), float (浮動小数点数)
  - str (文字列), bytes (バイナリ列)
  - コンテナ
  - 他にもあるけどとりあえずこれぐらい

Python のデータ型

- 組み込みデータ型
  - **int (整数)**, float (浮動小数点数)
  - str (文字列), bytes (バイナリ列)
  - **コンテナ**
  - 他にもあるけどとりあえずこれぐらい

int: struct _longobject

- 多倍長整数
  - (メモリの許す限り)いくらでも大きな整数を表すことができる

```
struct _longobject {
    PyVarObject ob_base;
    digit ob_digit[1];
};
```

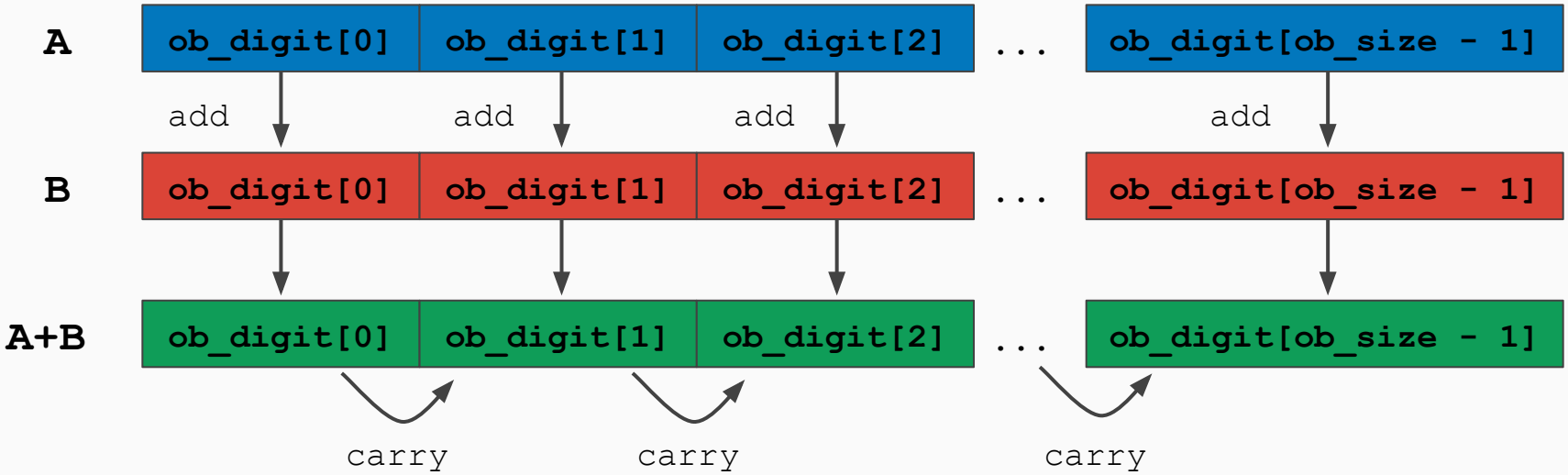
int: struct _longobject

- 負の数は ob_size を負にすることで表している
- 0 のときは ob_size == 0
- ある整数Xの絶対値に対して，次の関係が成り立つ
  - D_i は ob_digit[i]
  - N は ob_size の絶対値
  - S はシフト数 (15 もしくは 30)

$$|X| = \sum_{i=0}^{N-1} D_i \times 2^{S \times i}$$

多倍長整数の足し算・引き算

- 多倍長整数の足し算・引き算は，筆算に似た方法で行われる



多倍長整数の掛け算・割り算

- 多倍長整数の掛け算・割り算はちょっと複雑で説明しきれないので省略
  - 愚直にやると遅いので，最適化されたアルゴリズムが用いられている
  - 次のワードでググろう
    - カラツバ法
  - 次の書籍をあたろう
    - The Art of Computer Programming (D. Knuth)
    - Handbook of Applied Cryptography (Alfred J. Menezes ほか)

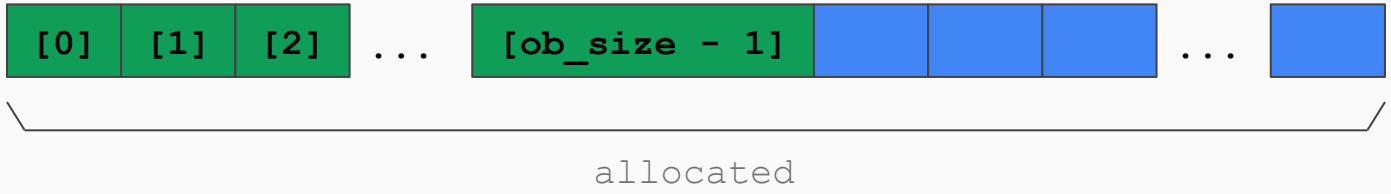
Python のコンテナ

- 次の4つを取り上げます
  - list
  - collections.deque
  - dict
  - set
- CPython では各種操作の時間計算量が規定されている
  - https://wiki.python.org/moin/TimeComplexity

list: PyListObject

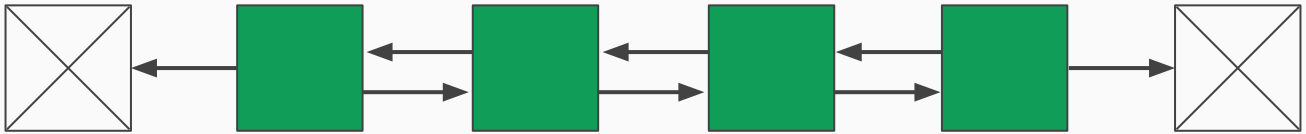
- list という名前をしているけど，動的配列

```
typedef struct {
    PyObject_VAR_HEAD      // PyVarObject としての情報
    PyObject **ob_item;    // 要素へのポインタ
    Py_ssize_t allocated; // 確保しているサイズ(≠要素数)
} PyListObject;
```



collections.deque

- 双方向連結リストによって実装された両端キュー
- 先頭への要素の追加がO(1)でできる
- 要素のローテート(Piet でいうところの roll)ができる



dict: PyDictObject

- ハッシュテーブル

```
typedef struct {
    PyObject_VAR_HEAD
    Py_ssize_t ma_used;  // 辞書の要素数
    uint64_t ma_version_tag; // 辞書のバージョン
    PyDictKeysObject *ma_keys; // 辞書のキーのリスト
    PyObject **ma_values; // 辞書の値を格納するテーブル
} PyDictObject;
```

dict: PyDictObject

- ハッシュテーブル

0

1

2

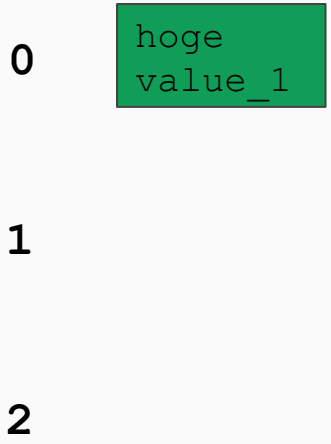
```
hoge
value_1
```

dict: PyDictObject

- ハッシュテーブル

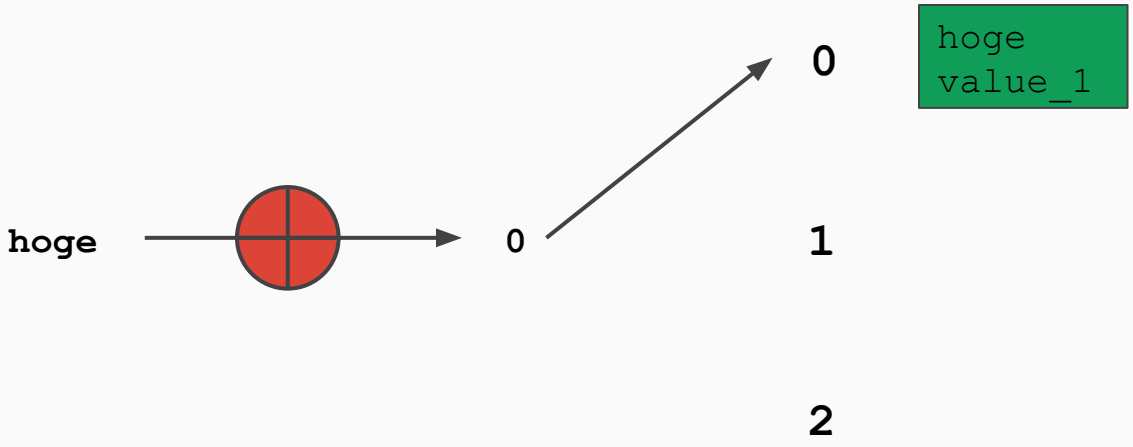
dict: PyDictObject

- ハッシュテーブル



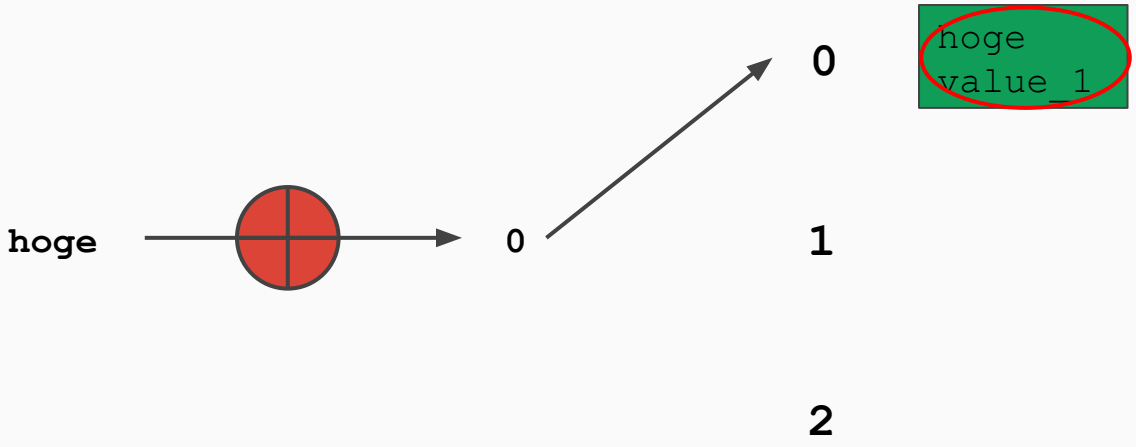
dict: PyDictObject

- ハッシュテーブル



dict: PyDictObject

- ハッシュテーブル

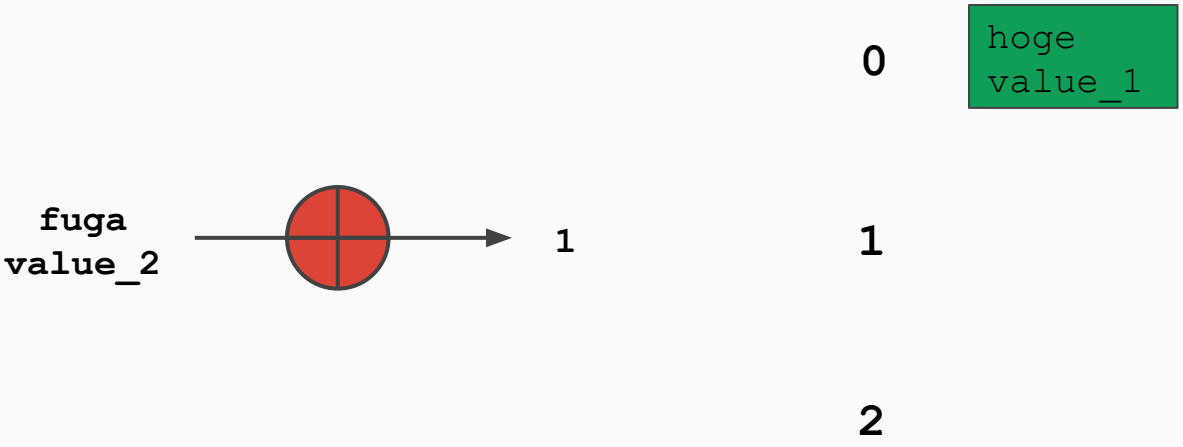


dict: PyDictObject

- ハッシュテーブル

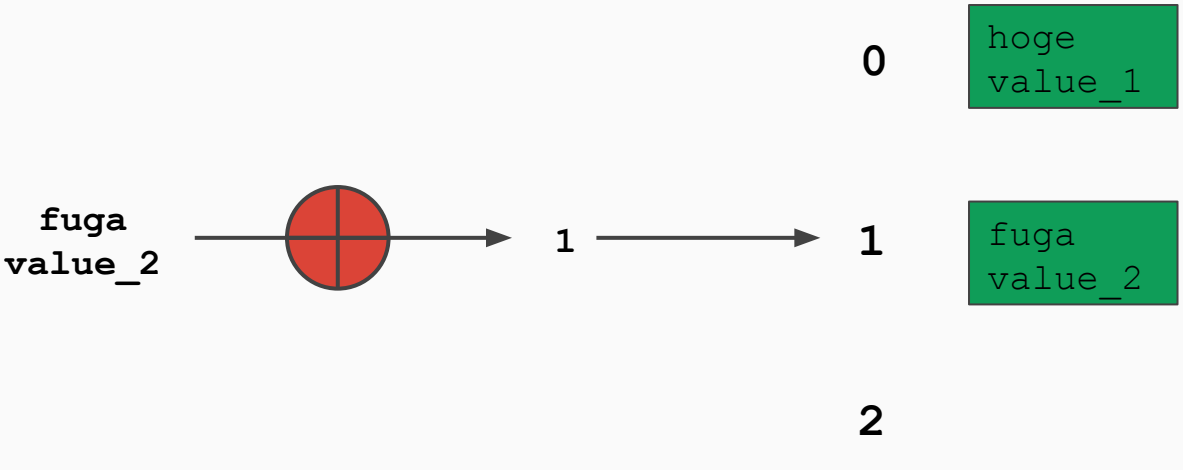
dict: PyDictObject

- ハッシュテーブル



dict: PyDictObject

- ハッシュテーブル



dict: PyDictObject

- ハッシュテーブル

piyo
value_3

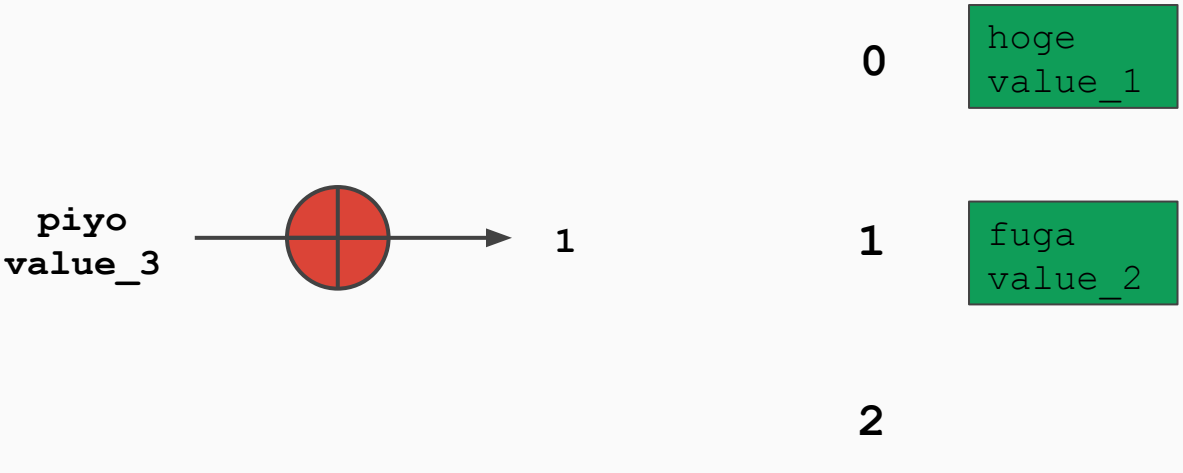
0 hoge value_1

1 fuga value_2

2

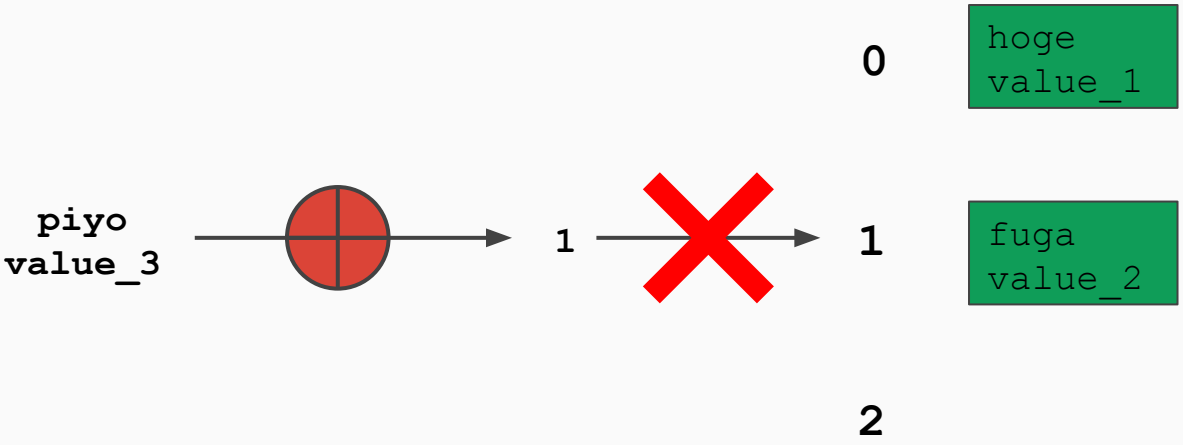
dict: PyDictObject

- ハッシュテーブル



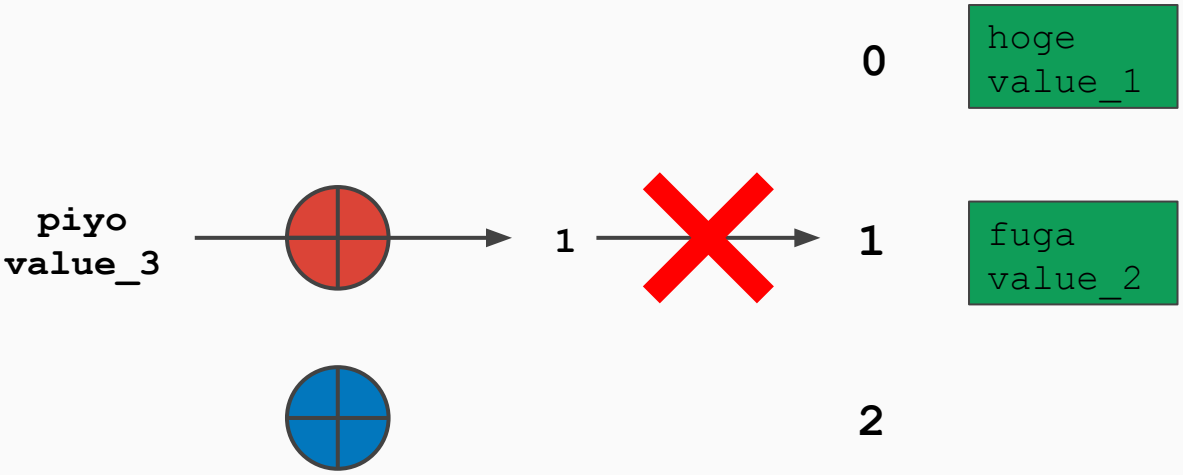
dict: PyDictObject

- ハッシュテーブル



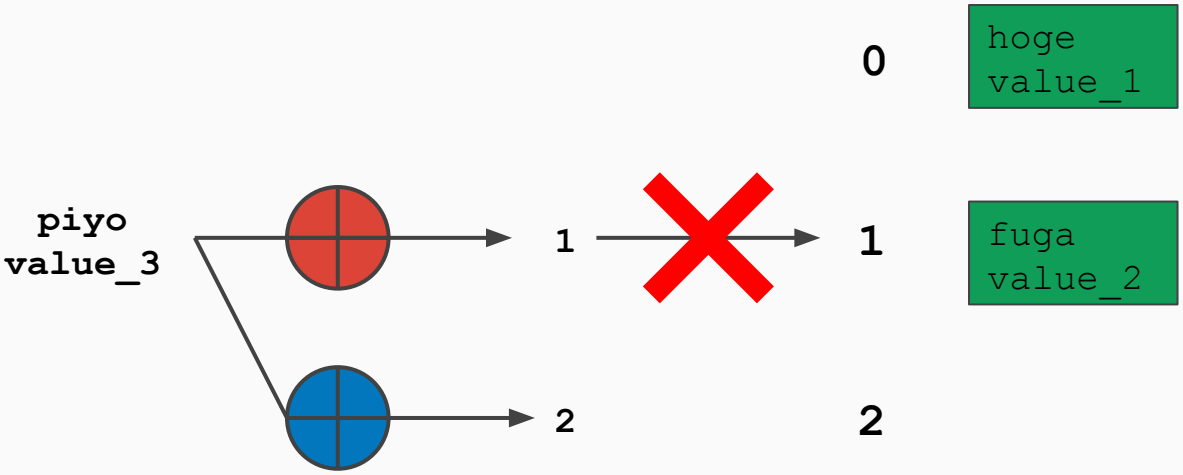
dict: PyDictObject

- ハッシュテーブル



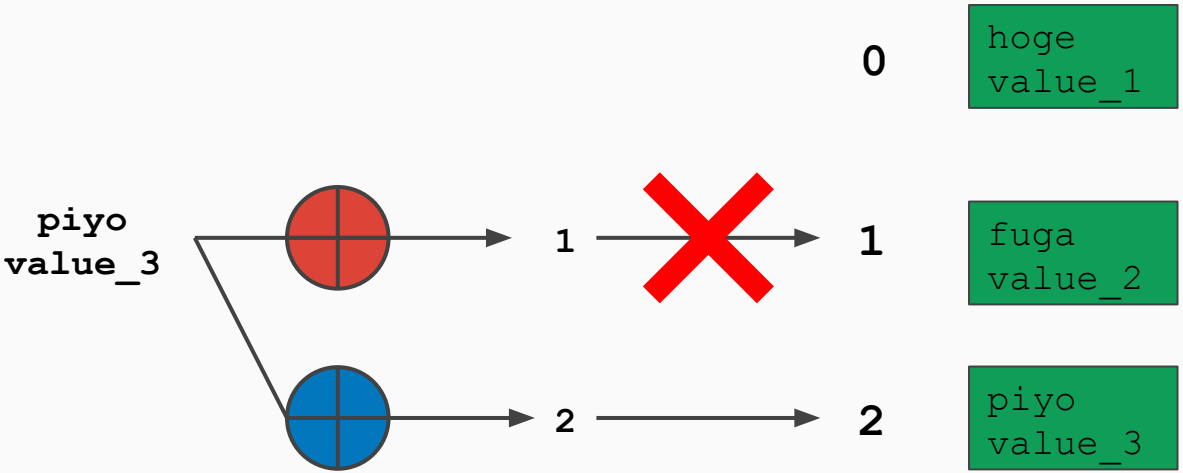
dict: PyDictObject

- ハッシュテーブル



dict: PyDictObject

- ハッシュテーブル



set

- (要素の重複を許さない)集合
- 実装はほとんど dict と同じなので省略
  - dict に key だけを格納するとだいたい set になる

Python の関数/モジュールの実装

モジュールの作り方

- だいたい以下の手順を踏めば CPython でライブラリが作れます
- クラスや定数を意図的に無視しています

1. 関数を書く
2. 関数の一覧を登録する
3. モジュールの情報を登録する
4. モジュールの初期化をする関数を書く

関数を書く

- 関数は `static` で宣言する
- 引数は `(PyObject*, PyObject*)` とする
- 返り値も `PyObject*` とする
- がんばる

関数を書く(具体的に)

- 関数は `static` で宣言する
  - 外部から見えていい関数は，後述する「モジュールを初期化する関数」のみとする
- 引数は `(PyObject*, PyObject*)` とする
  - 1つ目はモジュール自身を表す (だいたい self と名付ける)
  - 2つ目は引数を格納した Python のタプル (だいたい args と名付ける)
  - これを PyArg_ParseTuple() 関数を使ってパースすると，所望の値が得られる
- 返り値も `PyObject*` とする
  - None を返すときは Py_RETURN_NONE マクロを使う

関数を書く(具体的に)

- がんばる
  - あとはC言語で書くだけ！！！！！！！！
  - 手動で参照カウントを増減させる
    - このへんは Cython とか使うともっと楽に書けると思う
    - 今回はそこまで踏み込みません

関数の一覧を登録する

- PyMethodDef の配列に詰め込む
- 終端に番兵を置く

```
static PyMethodDef mysortmethods[] = {
  {“bubblesort”, (PyCFunction)bubblesort, METH_VARARGS, “bubblesort.”},
  {NULL, NULL, 0, NULL}
}
```

関数の一覧を登録する

- PyMethodDef の配列に詰め込む
- 終端に番兵を置く

```
static PyMethodDef mysortmethods[] = {
  {“bubblesort”, (PyCFunction)bubblesort, METH_VARARGS, “bubblesort.”},
  {NULL, NULL, 0, NULL}
}
```

関数の名前

関数の一覧を登録する

- PyMethodDef の配列に詰め込む
- 終端に番兵を置く

```
static PyMethodDef mysortmethods[] = {
  {“bubblesort”, (PyCFunction)bubblesort, METH_VARARGS, “bubblesort.”},
  {NULL, NULL, 0, NULL}
}
```

関数ポインタ
(PyCFunction にキャストする)

関数の一覧を登録する

- PyMethodDef の配列に詰め込む
- 終端に番兵を置く

```
static PyMethodDef mysortmethods[] = {
  {“bubblesort”, (PyCFunction)bubblesort, METH_VARARGS, “bubblesort.”},
  {NULL, NULL, 0, NULL}
}
```

関数の一覧を登録する

- PyMethodDef の配列に詰め込む
- 終端に番兵を置く

```
static PyMethodDef mysortmethods[] = {
  {“bubblesort”, (PyCFunction)bubblesort, METH_VARARGS, “bubblesort.”},
  {NULL, NULL, 0, NULL}
}
```

関数の一覧を登録する

- PyMethodDef の配列に詰め込む
- 終端に番兵を置く

```
static PyMethodDef mysortmethods[] = {
  {“bubblesort”, (PyCFunction)bubblesort, METH_VARARGS, “bubblesort.”},
  {NULL, NULL, 0, NULL}
}
```

番兵(終端を表す)

モジュールの情報を登録する

- PyModuleDef 構造体
  - モジュールの名前
  - モジュールの説明
  - モジュールが利用するメモリ領域の大きさ
  - モジュールの関数・クラス

モジュールの初期化をする関数を書く

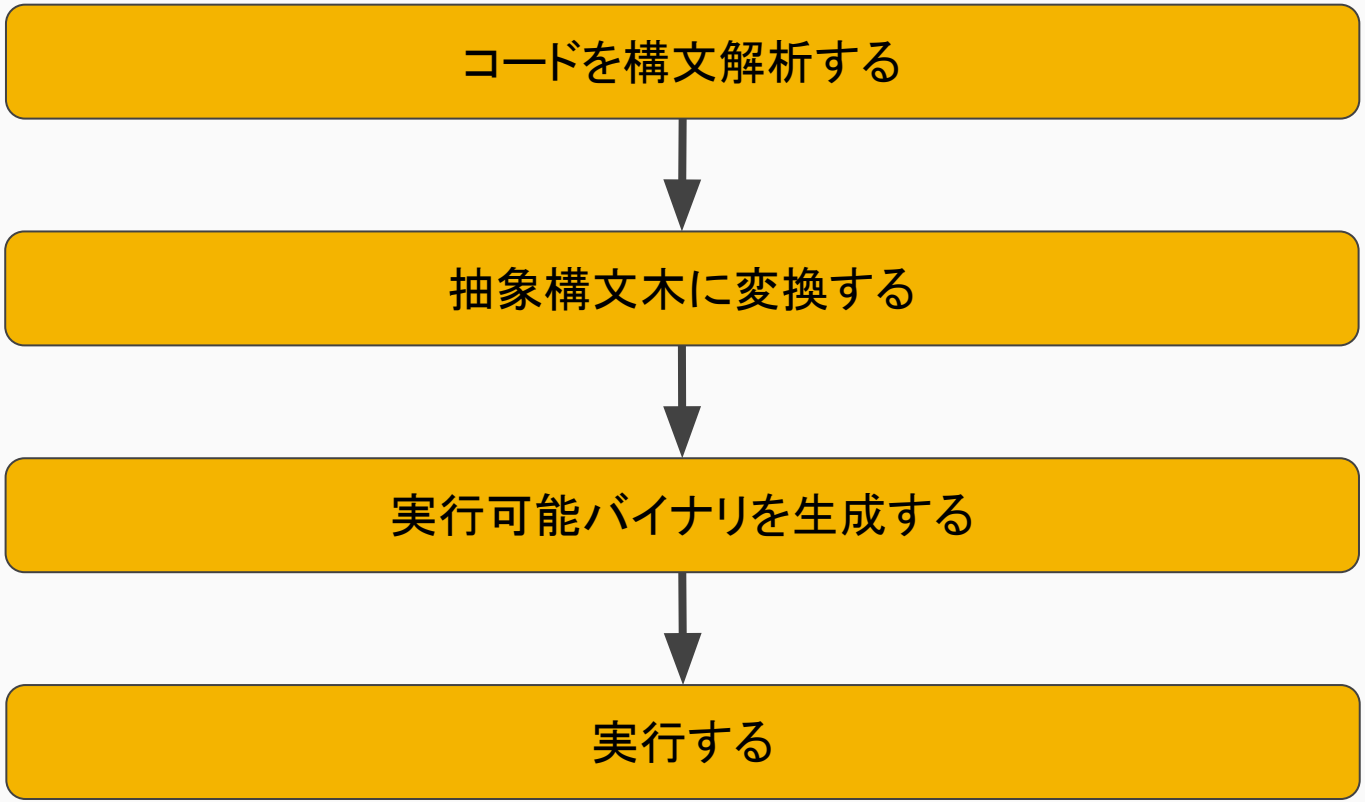
- これだけ非 static で宣言する
- これで import できるようになる

実例を見る

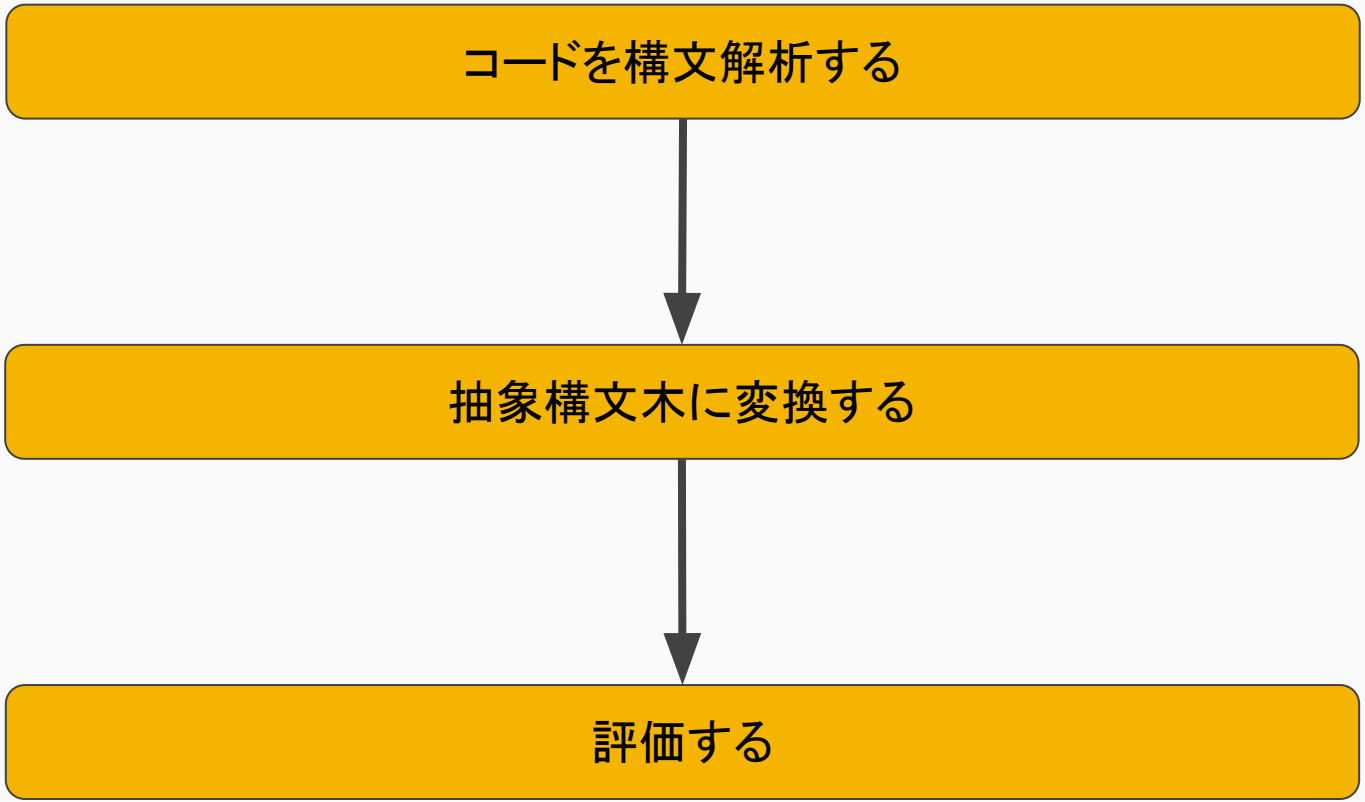
- バブルソート関数だけがある mysort というモジュールを書いた
  - https://github.com/utgwkk/python-c-api-practice
- setup.py をよしなに書くとモジュールとして配布できる
- 組み込みライブラリの場合は CPython のビルド時によしなにされる
- TODO: コードをざっと見る
- TODO: インストールできることを確認する

Python のコードの実行・仮想機械について

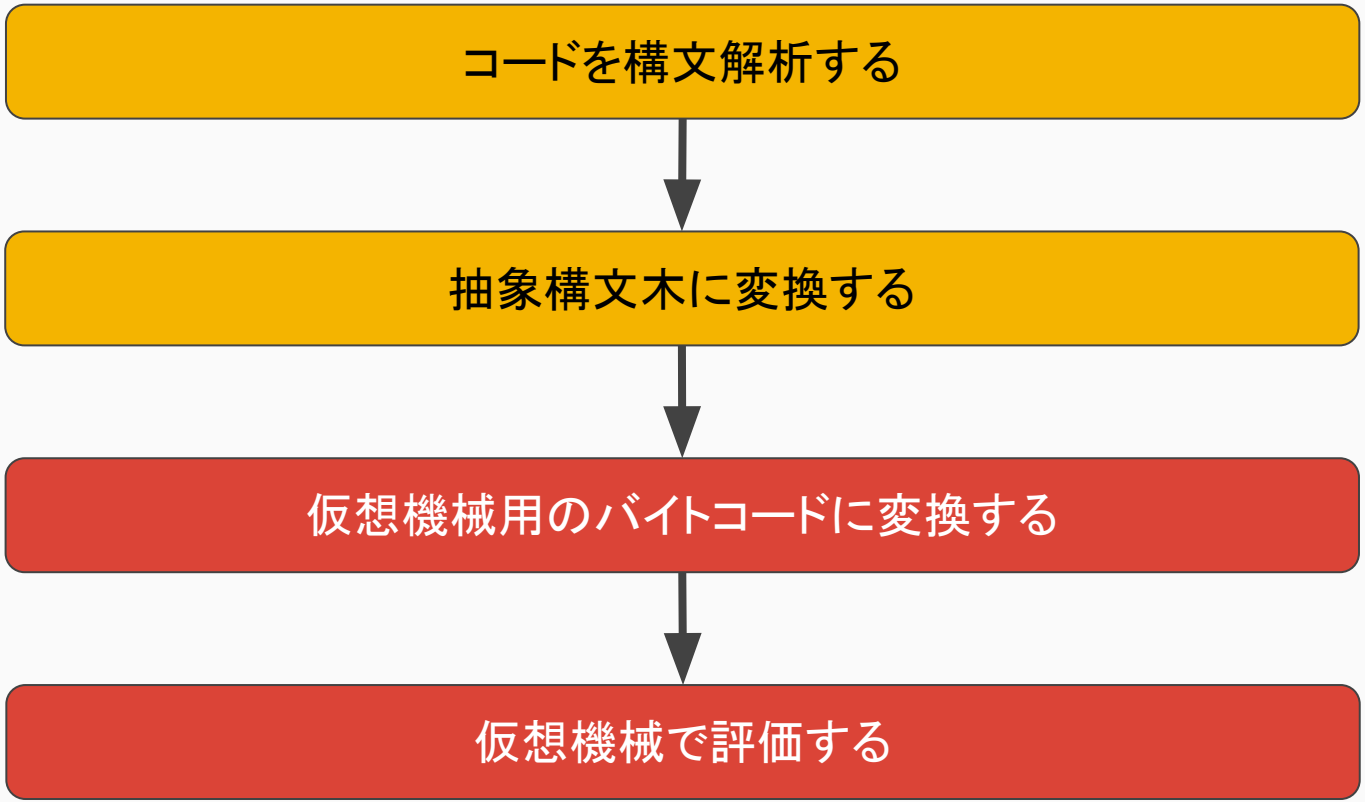
よくあるコンパイル言語の実行手順



素朴なインタプリタ言語の実行手順



Python の実行手順



実行の手順 (PEP 339 より)

1. コードを解析して解析木にする
2. 解析木を抽象構文木(AST)に変換する
3. 抽象構文木を制御フローグラフ(CFG)に変換する
4. 制御フローグラフを基にしてバイトコードを生成する
5. バイトコードを仮想機械上で実行する

実行の手順 (PEP 339 より)

1. コードを解析して解析木にする
2. 解析木を抽象構文木(AST)に変換する
3. 抽象構文木を制御フローグラフ(CFG)に変換する
4. 制御フローグラフを基にしてバイトコードを生成する
5. バイトコードを仮想機械上で実行する

バイトコードへの変換

- Python のコードは最終的にバイトコード(中間コード)にコンパイルされる
  - 1バイトで命令を表す (256通り)
  - これと引数を組み合わせて 2バイトが命令の最小単位
- コンパイルされたバイトコードは仮想機械(VM)上で実行される
- こういうタイプのインタプリタをバイトコードインタプリタと呼ぶ

なぜ「コンパイル」？

- 高級なコードを低級な命令(バイトコード)に変換してから
  仮想機械上で実行したほうが速い
- いろんな言語がこういう体系を取っている
  - Java, C#, Ruby, ...

PyCodeObject (Include/code.h)

- バイトコードを表す構造体
  - 引数の数
  - スタックの大きさ
  - 命令コード
  - ローカル変数の配列
  - 定数の配列

バイトコードを逆アセンブルする

- TODO:「Python のコードの例」で紹介したコードのバイトコードを見る
  - 条件分岐
  - 制御構造
  - 例外処理
  - コンテキストマネージャ
  - ジェネレータ
- TODO:見る
  - x, y = y, x

Python VM の仕様

- スタックマシン
  - データのやりとりにスタックを用いる
  - 計算対象としてスタックのトップから順に数えていく
- 命令一覧
  - http://docs.python.jp/3/library/dis.html#python-bytecode-instructions

Python VM の正体

- PyFrameObject 構造体がその正体
  - PyCodeObject へのポインタ(バイトコードの列)
  - グローバル変数, ローカル変数などへのポインタ
  - スタック
  - 例外オブジェクト, トレースバックの情報
- 関数呼び出しごとに作られる
  - ジェネレータが簡単に実現できる

_PyEval_EvalFrameDefault() 関数

- ここに PyFrameObject を渡して PyCodeObject を実行している
- 終了条件に達するまでバイトコードを順に実行する
  - 無限ループ (for (;;)) の中がその実体
  - 終了条件
    - return 文が実行される
    - 例外が送出され try〜except で捕捉されない

シミュレートしてみよう

```
1           0 LOAD_CONST               3 (6)
            2 STORE_NAME               0 (a)
            4 LOAD_NAME                1 (print)
            6 LOAD_NAME                0 (a)
            8 CALL_FUNCTION            1
           10 POP_TOP
           12 LOAD_CONST               2 (None)
           14 RETURN_VALUE
```

定数:
None
6

名前:
a
print

シミュレートしてみよう

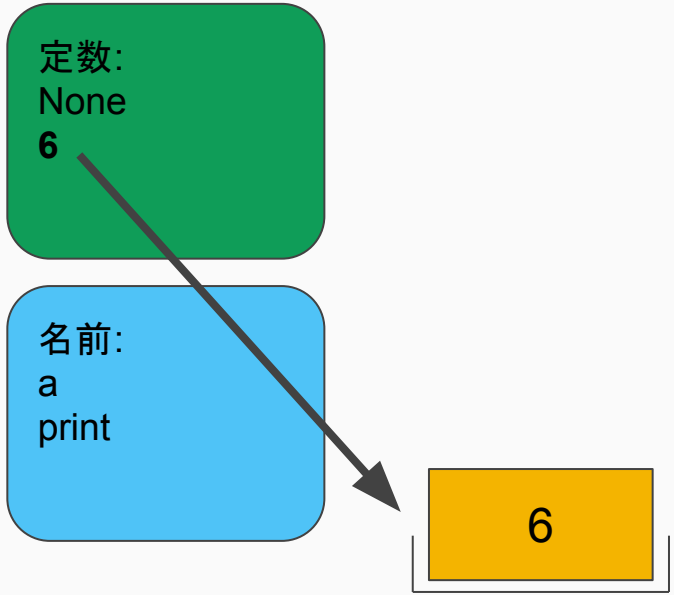
```
1           0 LOAD_CONST               3 (6)
            2 STORE_NAME               0 (a)
            4 LOAD_NAME                1 (print)
            6 LOAD_NAME                0 (a)
            8 CALL_FUNCTION            1
           10 POP_TOP
           12 LOAD_CONST               2 (None)
           14 RETURN_VALUE
```

定数:
None
6

名前:
a
print

シミュレートしてみよう

```
1           0 LOAD_CONST               3 (6)
            2 STORE_NAME               0 (a)
            4 LOAD_NAME                1 (print)
            6 LOAD_NAME                0 (a)
            8 CALL_FUNCTION            1
           10 POP_TOP
           12 LOAD_CONST               2 (None)
           14 RETURN_VALUE
```



シミュレートしてみよう

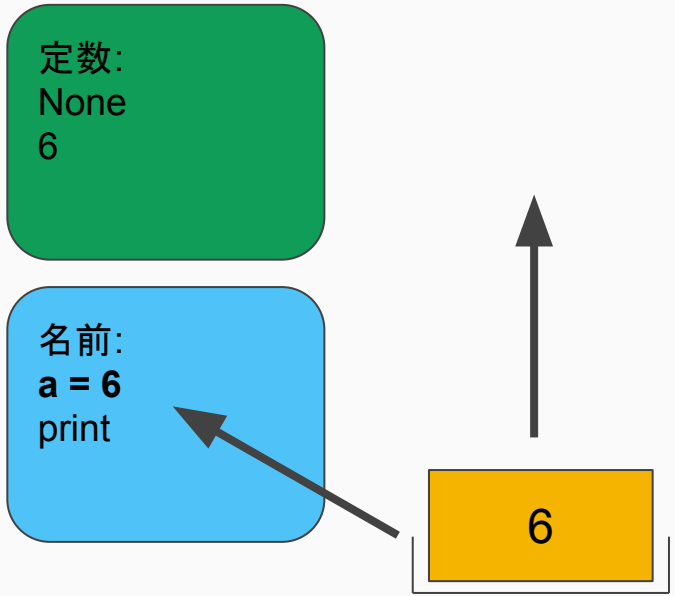
```
1      0 LOAD_CONST          3 (6)
       2 STORE_NAME          0 (a)
       4 LOAD_NAME           1 (print)
       6 LOAD_NAME           0 (a)
       8 CALL_FUNCTION       1
      10 POP_TOP
      12 LOAD_CONST          2 (None)
      14 RETURN_VALUE
```

定数:
None
6

名前:
a
print

シミュレートしてみよう

```
1      0 LOAD_CONST          3 (6)
       2 STORE_NAME          0 (a)
       4 LOAD_NAME           1 (print)
       6 LOAD_NAME           0 (a)
       8 CALL_FUNCTION       1
      10 POP_TOP
      12 LOAD_CONST          2 (None)
      14 RETURN_VALUE
```



シミュレートしてみよう

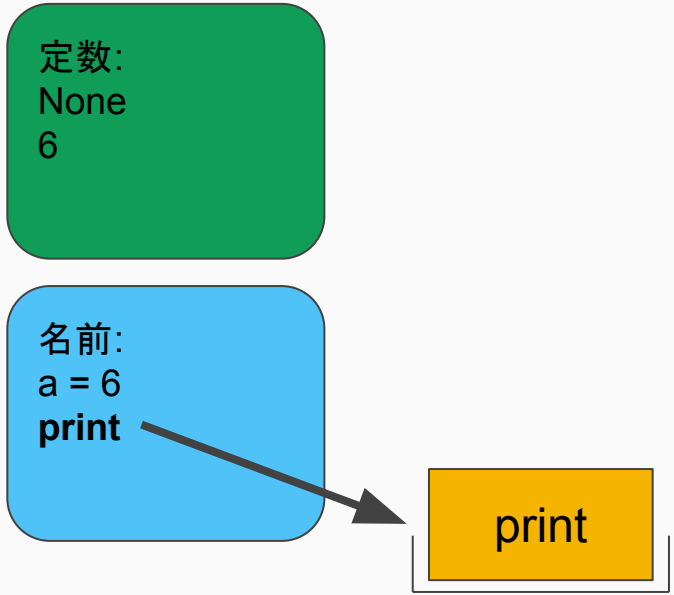
```
1       0 LOAD_CONST          3 (6)
        2 STORE_NAME          0 (a)
        4 LOAD_NAME           1 (print)
        6 LOAD_NAME           0 (a)
        8 CALL_FUNCTION       1
       10 POP_TOP
       12 LOAD_CONST          2 (None)
       14 RETURN_VALUE
```

定数:
None
6

名前:
a = 6
print

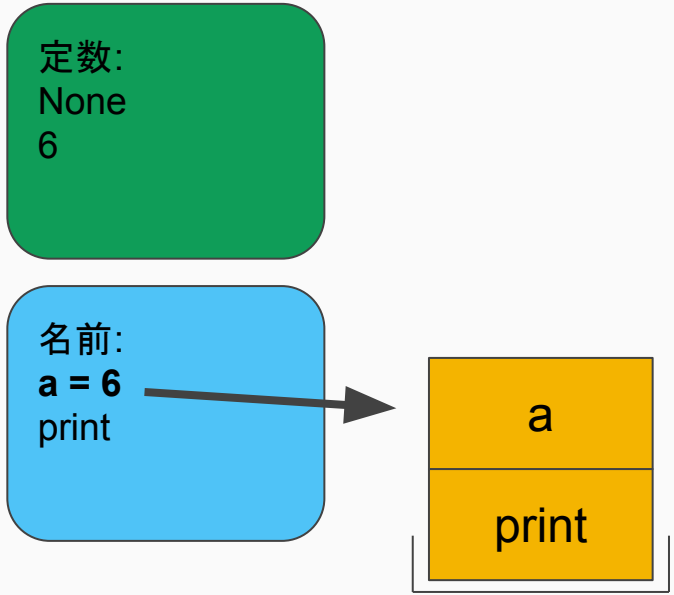
シミュレートしてみよう

```
1      0 LOAD_CONST          3 (6)
       2 STORE_NAME          0 (a)
       4 LOAD_NAME           1 (print)
       6 LOAD_NAME           0 (a)
       8 CALL_FUNCTION       1
      10 POP_TOP
      12 LOAD_CONST          2 (None)
      14 RETURN_VALUE
```



シミュレートしてみよう

```
1      0 LOAD_CONST          3 (6)
       2 STORE_NAME          0 (a)
       4 LOAD_NAME           1 (print)
       6 LOAD_NAME           0 (a)
       8 CALL_FUNCTION       1
      10 POP_TOP
      12 LOAD_CONST          2 (None)
      14 RETURN_VALUE
```

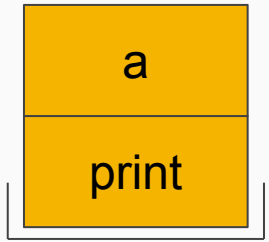


シミュレートしてみよう

```
1      0 LOAD_CONST          3 (6)
       2 STORE_NAME          0 (a)
       4 LOAD_NAME           1 (print)
       6 LOAD_NAME           0 (a)
       8 CALL_FUNCTION       1
      10 POP_TOP
      12 LOAD_CONST          2 (None)
      14 RETURN_VALUE
```

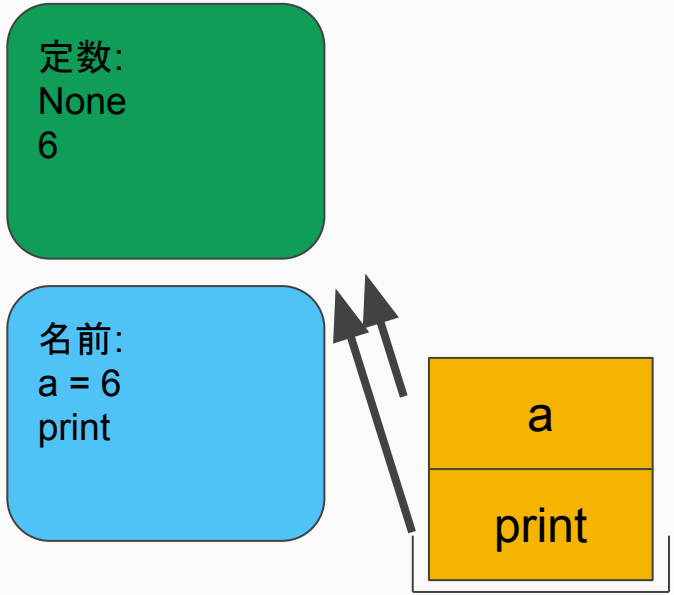
定数:
None
6

名前:
a = 6
print



シミュレートしてみよう

```
1       0 LOAD_CONST          3 (6)
        2 STORE_NAME          0 (a)
        4 LOAD_NAME           1 (print)
        6 LOAD_NAME           0 (a)
        8 CALL_FUNCTION       1
       10 POP_TOP
       12 LOAD_CONST          2 (None)
       14 RETURN_VALUE
```



シミュレートしてみよう

```
1      0 LOAD_CONST          3 (6)
       2 STORE_NAME          0 (a)
       4 LOAD_NAME           1 (print)
       6 LOAD_NAME           0 (a)
       8 CALL_FUNCTION       1
      10 POP_TOP
      12 LOAD_CONST          2 (None)
      14 RETURN_VALUE
```

定数:
None
6

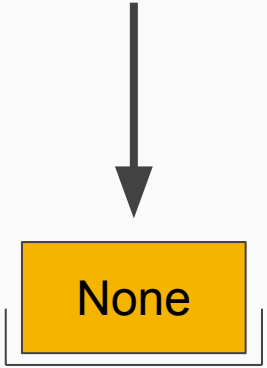
名前:
a = 6
print

シミュレートしてみよう

```
1       0 LOAD_CONST          3 (6)
        2 STORE_NAME          0 (a)
        4 LOAD_NAME           1 (print)
        6 LOAD_NAME           0 (a)
        8 CALL_FUNCTION       1
       10 POP_TOP
       12 LOAD_CONST          2 (None)
       14 RETURN_VALUE
```

定数:
None
6

名前:
a = 6
print



シミュレートしてみよう

```
1       0 LOAD_CONST          3 (6)
        2 STORE_NAME          0 (a)
        4 LOAD_NAME           1 (print)
        6 LOAD_NAME           0 (a)
        8 CALL_FUNCTION       1
       10 POP_TOP
       12 LOAD_CONST          2 (None)
       14 RETURN_VALUE
```

定数:
None
6

名前:
a = 6
print

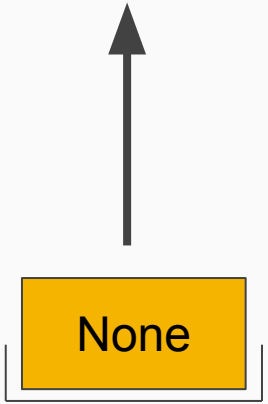
None

シミュレートしてみよう

```
1       0 LOAD_CONST          3 (6)
        2 STORE_NAME          0 (a)
        4 LOAD_NAME           1 (print)
        6 LOAD_NAME           0 (a)
        8 CALL_FUNCTION       1
       10 POP_TOP
       12 LOAD_CONST          2 (None)
       14 RETURN_VALUE
```

定数:
None
6

名前:
a = 6
print



シミュレートしてみよう

```
1      0 LOAD_CONST          3 (6)
       2 STORE_NAME          0 (a)
       4 LOAD_NAME           1 (print)
       6 LOAD_NAME           0 (a)
       8 CALL_FUNCTION       1
      10 POP_TOP
      12 LOAD_CONST          2 (None)
      14 RETURN_VALUE
```

定数:
None
6

名前:
a = 6
print

シミュレートしてみよう

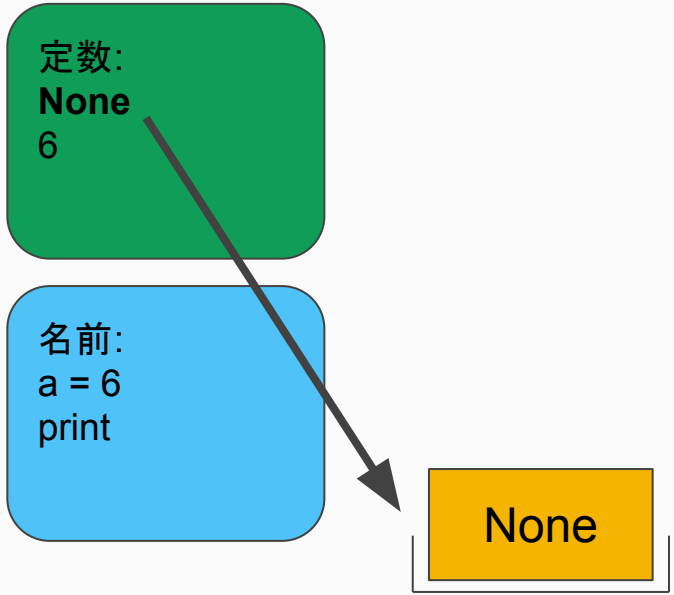
```
1      0 LOAD_CONST          3 (6)
       2 STORE_NAME          0 (a)
       4 LOAD_NAME           1 (print)
       6 LOAD_NAME           0 (a)
       8 CALL_FUNCTION       1
      10 POP_TOP
      12 LOAD_CONST          2 (None)
      14 RETURN_VALUE
```

定数:
None
6

名前:
a = 6
print

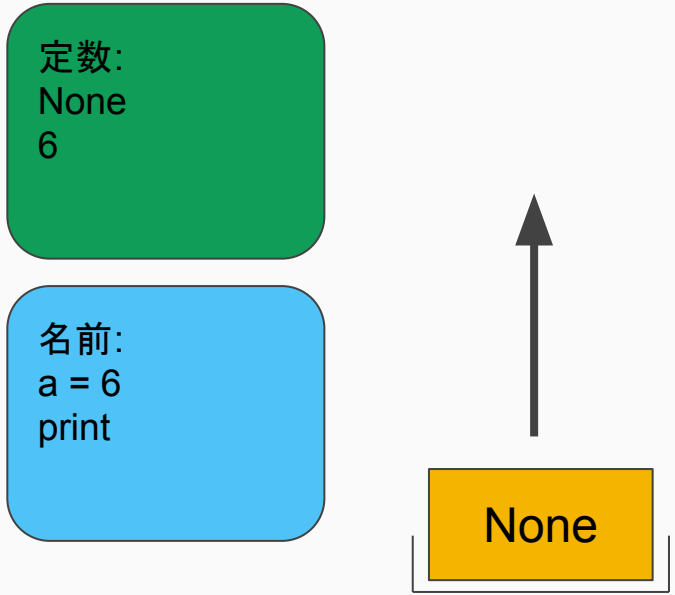
シミュレートしてみよう

```
1       0 LOAD_CONST          3 (6)
        2 STORE_NAME          0 (a)
        4 LOAD_NAME           1 (print)
        6 LOAD_NAME           0 (a)
        8 CALL_FUNCTION       1
       10 POP_TOP
       12 LOAD_CONST          2 (None)
       14 RETURN_VALUE
```



シミュレートしてみよう

```
1           0 LOAD_CONST               3 (6)
            2 STORE_NAME               0 (a)
            4 LOAD_NAME                1 (print)
            6 LOAD_NAME                0 (a)
            8 CALL_FUNCTION            1
           10 POP_TOP
           12 LOAD_CONST               2 (None)
           14 RETURN_VALUE
```



シミュレートしてみよう

```
1      0 LOAD_CONST          3 (6)
       2 STORE_NAME          0 (a)
       4 LOAD_NAME           1 (print)
       6 LOAD_NAME           0 (a)
       8 CALL_FUNCTION       1
      10 POP_TOP
      12 LOAD_CONST          2 (None)
      14 RETURN_VALUE
```

定数:
None
6

名前:
a = 6
print

まとめ

まとめ

- Python のオブジェクトはいろいろな構造体で表現されている
- Python のコードは最適化を経てバイトコードにコンパイルされる
- Python のコンパイルされたバイトコードは VM 上で実行される
- 質問は分かる範囲で答えます

参考

- Inside The Python Virtual Machine
  https://leanpub.com/insidethepythonvirtualmachine/read
- About Python VM
  http://svn.coderepos.org/share/docs/jbking/vmdoc/about_python_vm.html
- CPython の Git リポジトリ
  https://github.com/python/cpython
- Python 3.6.0 Documentation
  http://docs.python.jp/3/