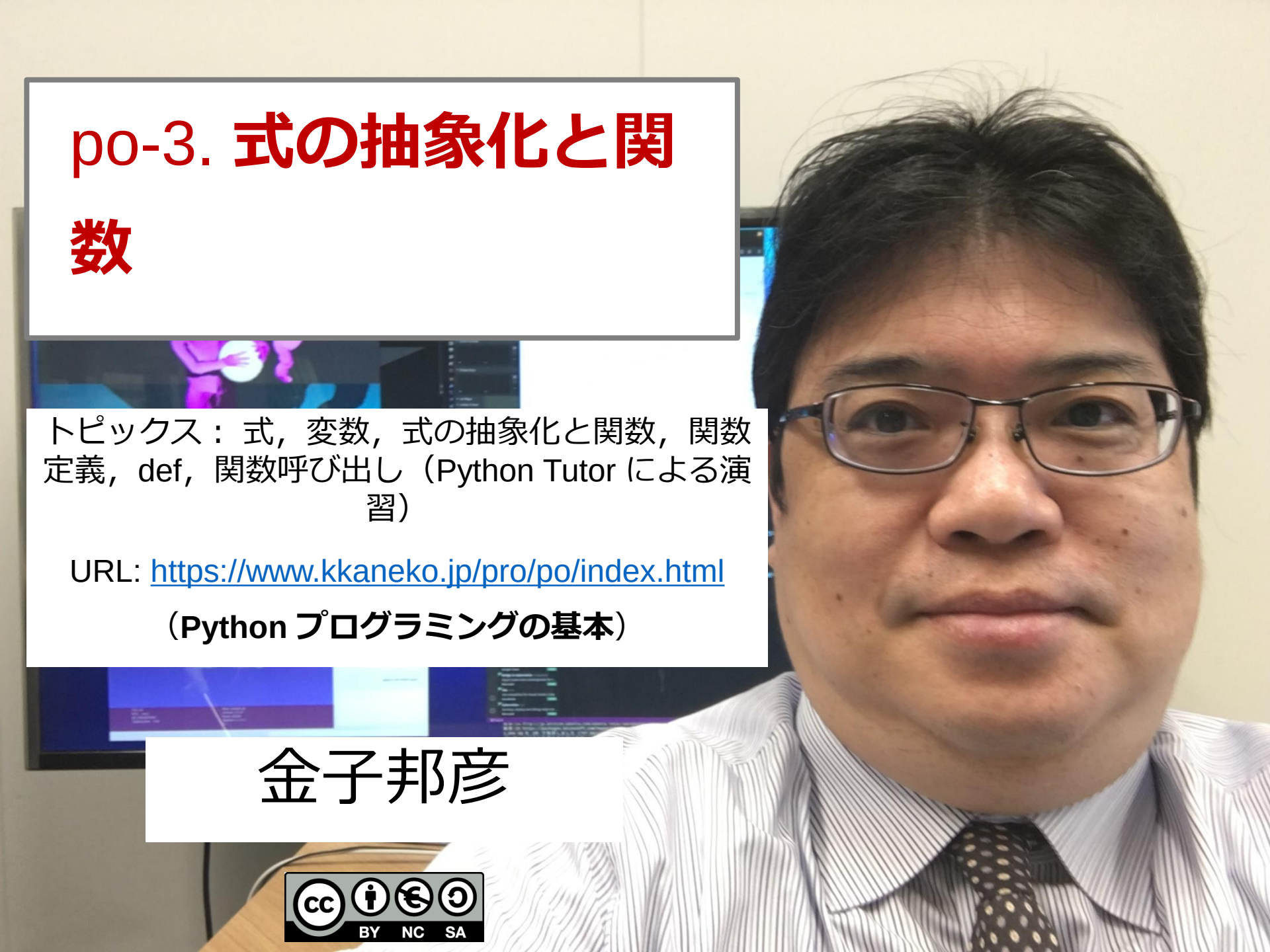
po-3. 式の抽象化と関数



トピックス： 式，変数，式の抽象化と関数，関数定義，def，関数呼び出し（Python Tutor による演習）

URL: https://www.kkaneko.jp/pro/po/index.html

（**Python プログラミングの基本**）

金子邦彦

式の抽象化と関数

```
1  print(100 * 1.1)
2  print(150 * 1.1)
3  print(200 * 1.1)
```

プログラム

Print output (drag lower right corner to resize)

```
110.00000000000001
165.0
220.00000000000003
```

Frames　　Objects

実行結果

```
1  def foo(a):
2      return a * 1.1
3  print(foo(100))
4  print(foo(150))
5  print(foo(400))
```

Print output (drag lower rigl

```
110.00000000000001
165.0
440.00000000000006
```

実行結果

３つの式を１つにまとめる

式「**a * 1.1**」を含む
関数 foo を**定義**

110.00000000000001
440.00000000000006
は計算誤差を含む
（誤作動やミスではない）

アウトライン

	項目
	復習
3-1	式，変数
3-2	式の抽象化と関数，関数定義，def
3-3	演習

ソースコード (source code)

- **プログラム**を，何らかのプログラミング言語で書いたもの
- 「**ソフトウエアの設計図**」ということも.
- **人間も読み書き，編集できる**

```
import picamera
camera = picamera.PiCamera()
camera.capture("1.jpg")
exit()
```

Raspberry Pi で，カメラを使って
撮影し，画像を保存するプログラムの
ソースコード（Python 言語）

オブジェクト，変数，メソッド，代入，変数

- **オブジェクト**：コンピュータでの**操作や処理の対象となるもの**のこと
- **変数**：**名前の付いたオブジェクト**には，**変数**，**関数**，**モジュール**などがある（変数や関数は，数学の変数や関数とは**違う意味**）

- **メソッド: オブジェクト**に属する操作や処理．**メソッド**呼び出しでは，**引数**を指定することがある．**引数**（ひきすう）は，**メソッド**に渡す値のこと

```
Hero.attack("fence", 36, 26)
```

- **代入**：「=」を使用．オブジェクトの**値が変化**する

```
b = a + 100
```

メソッドアクセス，代入

Python プログラムの例

```
x = 100
a = x + 200
enermy1 = hero.findNearestEnemy()
hero.attack(enemy1)
```

- **代入**：**オブジェクト名** ＋ 「**=**」
  ＋ 式または値またはメソッド呼び出し
- **メソッドアクセス**：**オブジェクト名** ＋ 「**.**」
  ＋ **メソッド名** ＋「**()**」 （引数を付けることも）

Python プログラムでは，その他にも，属性アクセス，関数呼び出し，制御，「*」，「+」などの演算子，コマンド，定義など

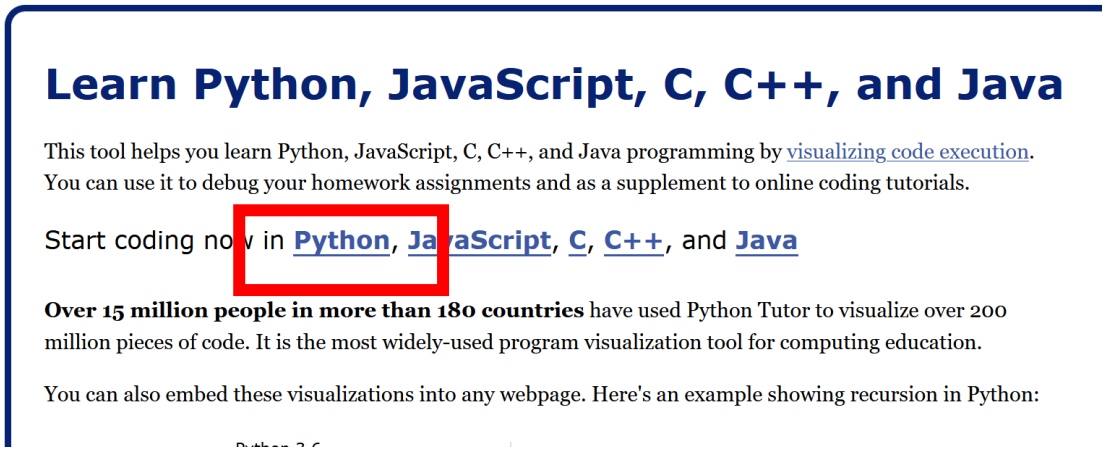
Python Tutor の起動

① **ウェブブラウザ**を起動する

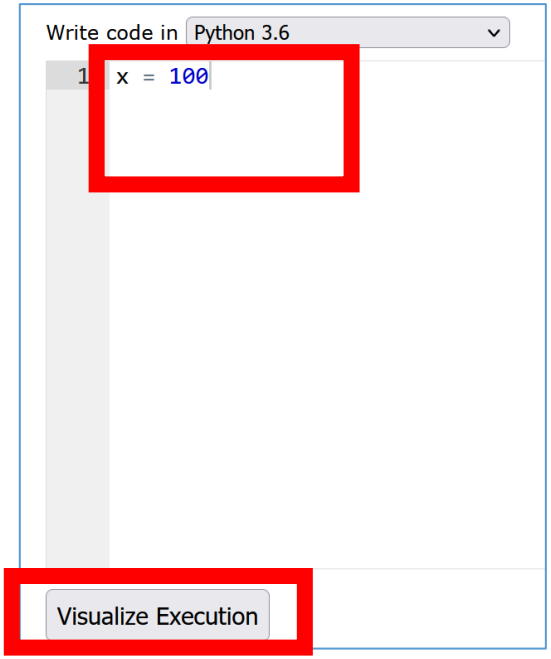
② **Python Tutor** を使いたいので，次の URL を開く

http://www.pythontutor.com/

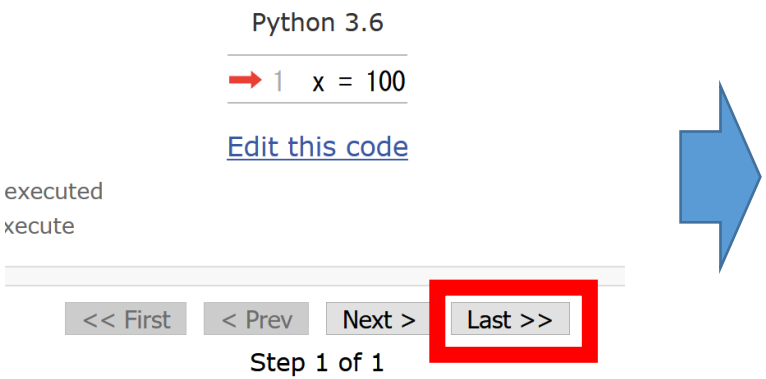
③ 「**Python**」をクリック　⇒ **編集画面**が開く



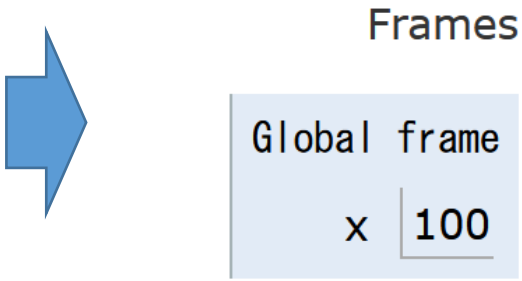
Python Tutor でのプログラム実行手順



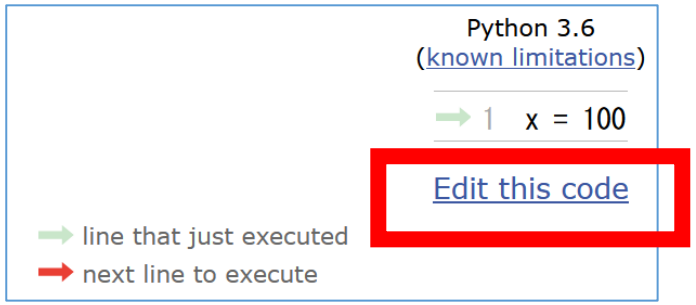
(1)「**Visualize Execution**」をクリックして**実行画面**に切り替える



(2)「**Last**」をクリック.



(3) **<u>実行結果を確認</u>**する.



(4)「**Edit this code**」をクリックして**編集画面**に戻る

Python Tutor 使用上の注意点①

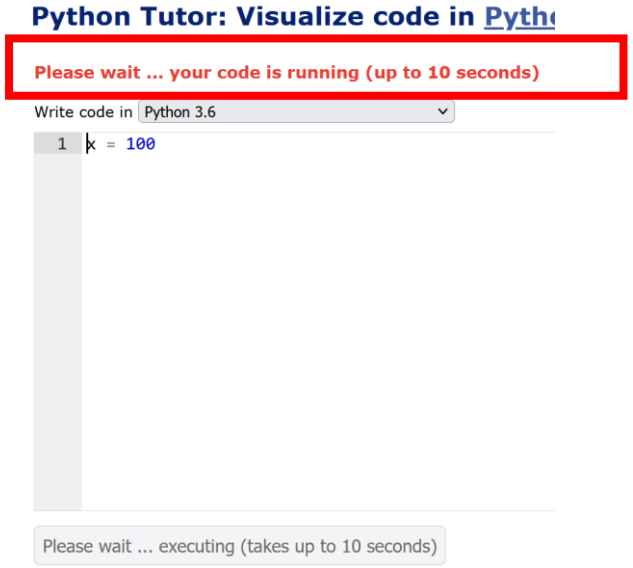
- **実行画面で，次のような赤の表示が出ることがある → 無視してよい**

  過去の文法ミスに関する確認表示

  邪魔なときは「**Close**」

Python Tutor 使用上の注意点②

「**please wait ... executing**」のとき，**１０秒ほど待つ**.



→　**混雑しているとき**は，**「Server Busy・・・」というメッセージが出る**ことがある.

混雑している．**少し（数秒から数十秒）待つ**と自動で表示が変わる（変わらない場合には，操作をもう一度行ってみる）

3-1. 式，变数

いまから
行うこと

- Python で，**式**や**変数**や計算に上達する

式の実行結果

式の実行結果として，**値が得られる**

```
print(100 * 200)
```

```
Print output (drag
20000
```

プログラム　　**実行結果**

Python の変数

- **変数**：**名前の付いたオブジェクト**には，**変数**，**関数**などがある（変数や関数は，数学の変数や関数とは**違う意味**）
- **変数**は，「**値をコンピュータに覚えさせておくもの**」として使うことができる

```
x = 100
print(x)
```

```
100
```

```
y = 2000
print(y)
```

```
2000
```

変数には，値を代入できる

式

- **式**から値が求まる（コンピュータを使って，計算などができる）
- **式**は**変数**を含むことができる

```
print(10 + 20 + 30)
```

```
60
```

```
a = 100
print(a * 1.08)
```

```
108.0
```

代入

- **代入**：プログラムで，「**x = 100**」のように書くと，**x の値が 100 に変化**する

```
x = 100
```

Frames

Global frame

x	100

プログラム

実行結果

演習

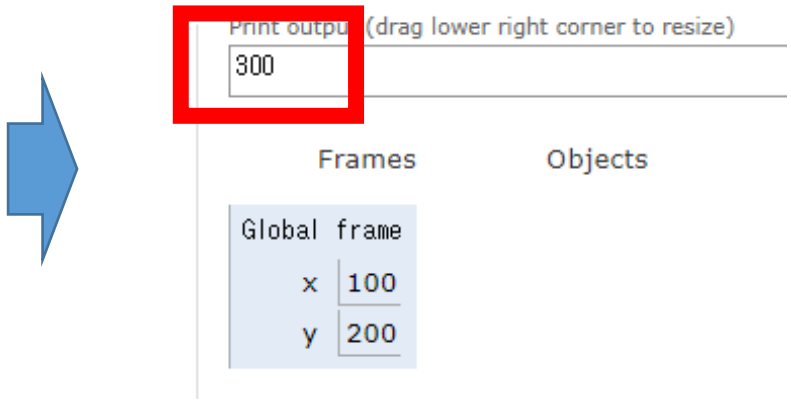
資料：**18 ～ 21**

【トピックス】

- **式と変数**

① Python Tutor のエディタで次のプログラムを入れ，実行し，結果を確認する

```
x = 100
y = 200
print(x + y)
```

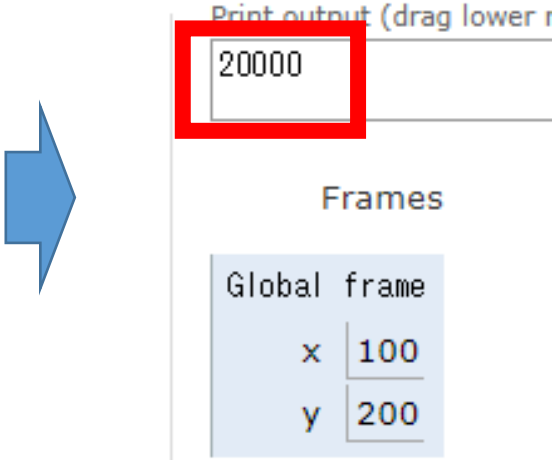


結果の
「**300**」を確認

「**Visual Execution**」をクリック．そして「**Last**」をクリック．結果を確認．
「**Edit this code**」をクリックすると，エディタの画面に戻る

② Python Tutor のエディタで次のプログラムを入れ，実行し，結果を確認する

```
x = 100
y = 200
print(x * y)
```

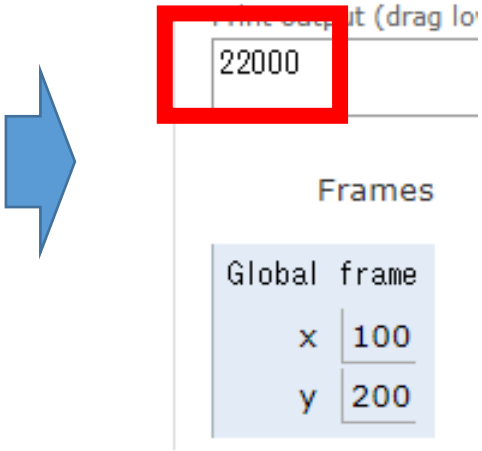


結果の
「**20000**」を確認

「**Visual Execution**」をクリック．そして「**Last**」をクリック．結果を確認．
「**Edit this code**」をクリックすると，エディタの画面に戻る

③ Python Tutor のエディタで次のプログラムを入れ，実行し，結果を確認する

```
x = 100
y = 200
print((x + 10) * y)
```



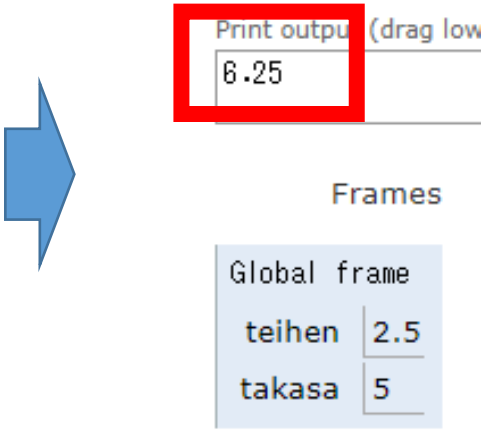
結果の
「**22000**」を確認

「**Visual Execution**」をクリック．そして「**Last**」をクリック．結果を確認．
「**Edit this code**」をクリックすると，エディタの画面に戻る

④ Python Tutor のエディタで次のプログラムを入れ，実行し，結果を確認する

底辺が**2．5**で，高さが**5**のとき，
三角形の面積は，面積：**6．25**

```
teihen = 2.5
takasa = 5
print(teihen * takasa / 2)
```



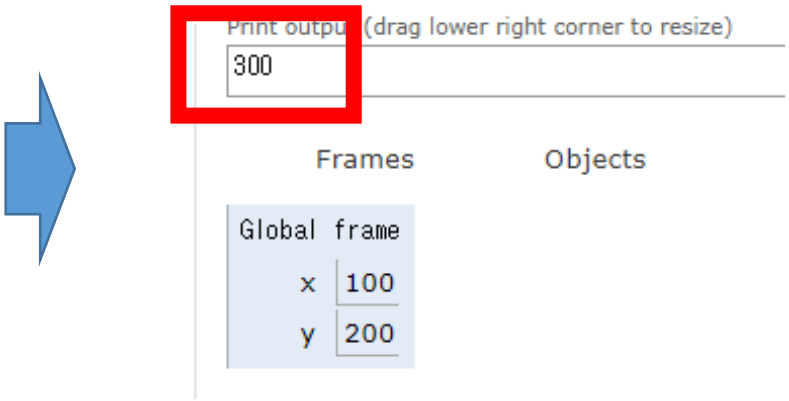
結果の
「**6.25**」を確認

「**Visual Execution**」をクリック．そして「**Last**」をクリック．結果を確認．
「**Edit this code**」をクリックすると，エディタの画面に戻る

まとめ

- **代入**：プログラムで，「**x = 100**」のように書くと，**x の値が 100 に変化**する
- **式**の実行結果として，**値が得られる**
- **式**の中に，**変数名**を書くことができる

```
x = 100
y = 200
print(x + y)
```



3-2. 式の抽象化と関数，関数定義，def

関数定義

```
def foo(a):
    return a * 1.1
```

- この**関数**の**本体**は
「**return a * 1.1**」

- この**関数**は，式「**a * 1.1**」に，**名前 foo** を付けたものと考えることもできる

関数

```
def foo(a):
  return a * 1.1
```

- この**関数**の**本体**は
  「**return a * 1.1**」

- この**関数**は，式「**a * 1.1**」に，**名前 foo** を付けたものと考えることもできる

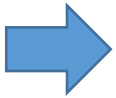
式の抽象化

100 * 1.1

150 * 1.1

400 * 1.1

類似した複数の**式**



a * 1.1

変数 a を使って，複数の**式**を１つにまとめる（**抽象化**）

式

100 * 1.1
150 * 1.1
400 * 1.1

変数を含む式

a * 1.1

関数

```
def foo(a):
    return a * 1.1
```

この**関数**は，式「**a * 1.1**」に，**名前 foo** を付けたものと考えることもできる

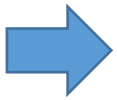
関数

100 * 1.1

150 * 1.1

400 * 1.1

類似した複数の**式**



a * 1.1

変数 a を使って，複数の**式**を１つにまとめる（**抽象化**）

```
def foo(a):
  return a * 1.1
print(foo(100))
print(foo(150))
print(foo(400))
```

式「**a * 1.1**」を含む
関数 foo を**定義**

関数 foo を使用.
100, 150, 400 は**引数**

```
110.00000000000001
165.0
440.00000000000006
```

式の抽象化と関数

抽象化前

```
print(100 * 1.1)
print(150 * 1.1)
print(400 * 1.1)
```

類似した複数の**式**

```
110.00000000000001
165.0
440.00000000000006
```

実行結果

抽象化後

```
def foo(a):
    return a * 1.1
print(foo(100))
print(foo(150))
print(foo(400))
```

関数の定義と使用

```
110.00000000000001
165.0
440.00000000000006
```

同じ
実行結果になる

抽象化がなぜ大切なのか

- プログラミングでの根本問題は何でしょうか？
  - **誤り（バグ）の無いプログラムの作成**

- プログラミングの一番の基礎は何でしょうか？
  - **抽象化**を行うこと.
  - **抽象化**により，**繰り返し同じことを書くことが減り，バグを防げる**.
  - プログラムの変更も簡単に.

演習

資料：**32**

【トピックス】

- **式の抽象化と関数**
- **関数定義**
- **def**

Python Tutor のエディタで次のプログラムを入れ，実行し，結果を確認する

```
def foo(a):
    return a * 1.1
print(foo(100))
print(foo(150))
print(foo(400))
```

Print output (drag lower right

```
110.00000000000001
165.0
440.00000000000006
```

結果を確認

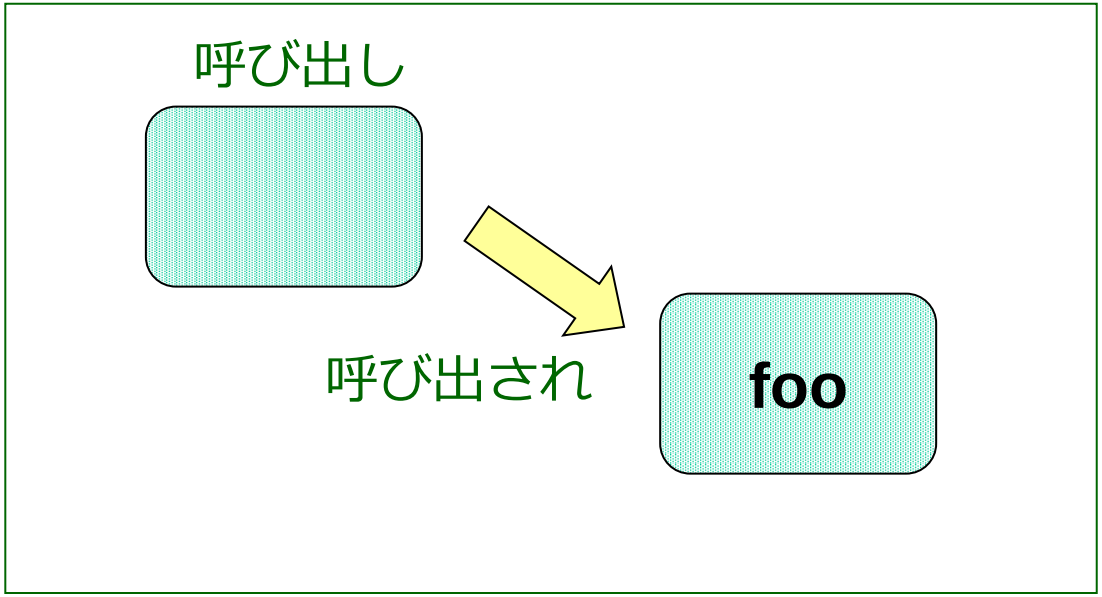
（計算誤差がある．
動作は正常）

「return a * 1.1」の行は字下げが必要

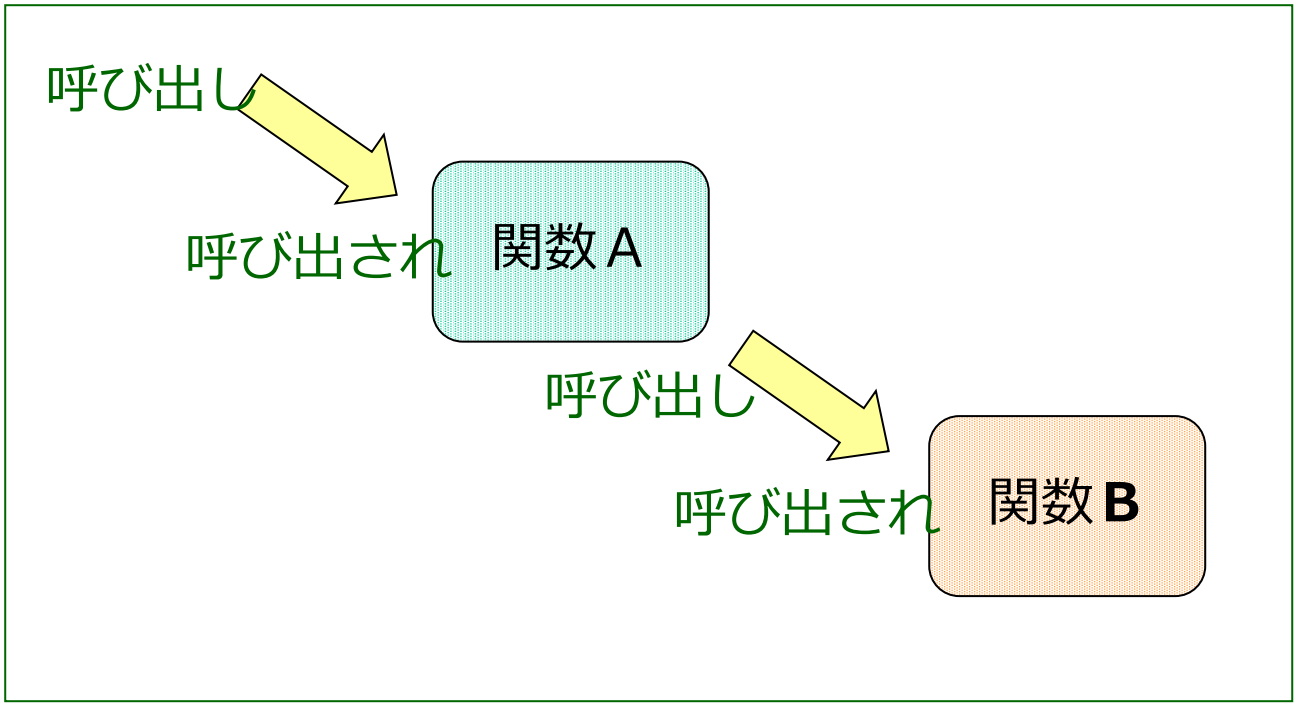
「**Visual Execution**」をクリック．そして「**Last**」をクリック．結果を確認．
「**Edit this code**」をクリックすると，エディタの画面に戻る

3-3. 関数呼び出し

関数呼び出し



関数呼び出しの例



プログラムは，しばしば，複数の
関数に「分割」される

- 関数の中で関数を呼び出す場合

```
def foo(a):
    return a * 1.08
def bar(x):
    return foo(x) * 100
p = 12
print(bar(p))
p = 20
print(bar(p))
```

foo の呼び出し

bar の呼び出し

ステップ実行

ステップ実行により、プログラム実行の流れをビジュアルに観察

演習

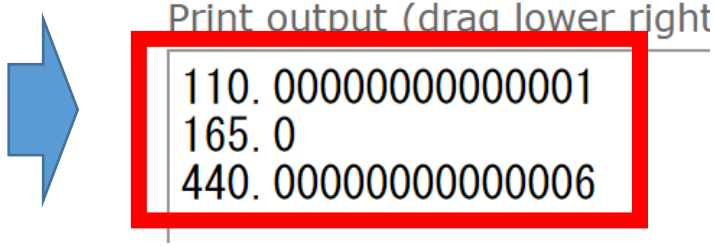
資料：**39 ～ 45**

【トピックス】

- **関数呼び出し**における**ジャンプ**
- **関数**内で使用される変数が**消える**タイミング

① Python Tutor のエディタで次のプログラムを入れ，実行し，結果を確認する（あとで使うので消さないこと）

```
def foo(a):
    return a * 1.1
print(foo(100))
print(foo(150))
print(foo(400))
```



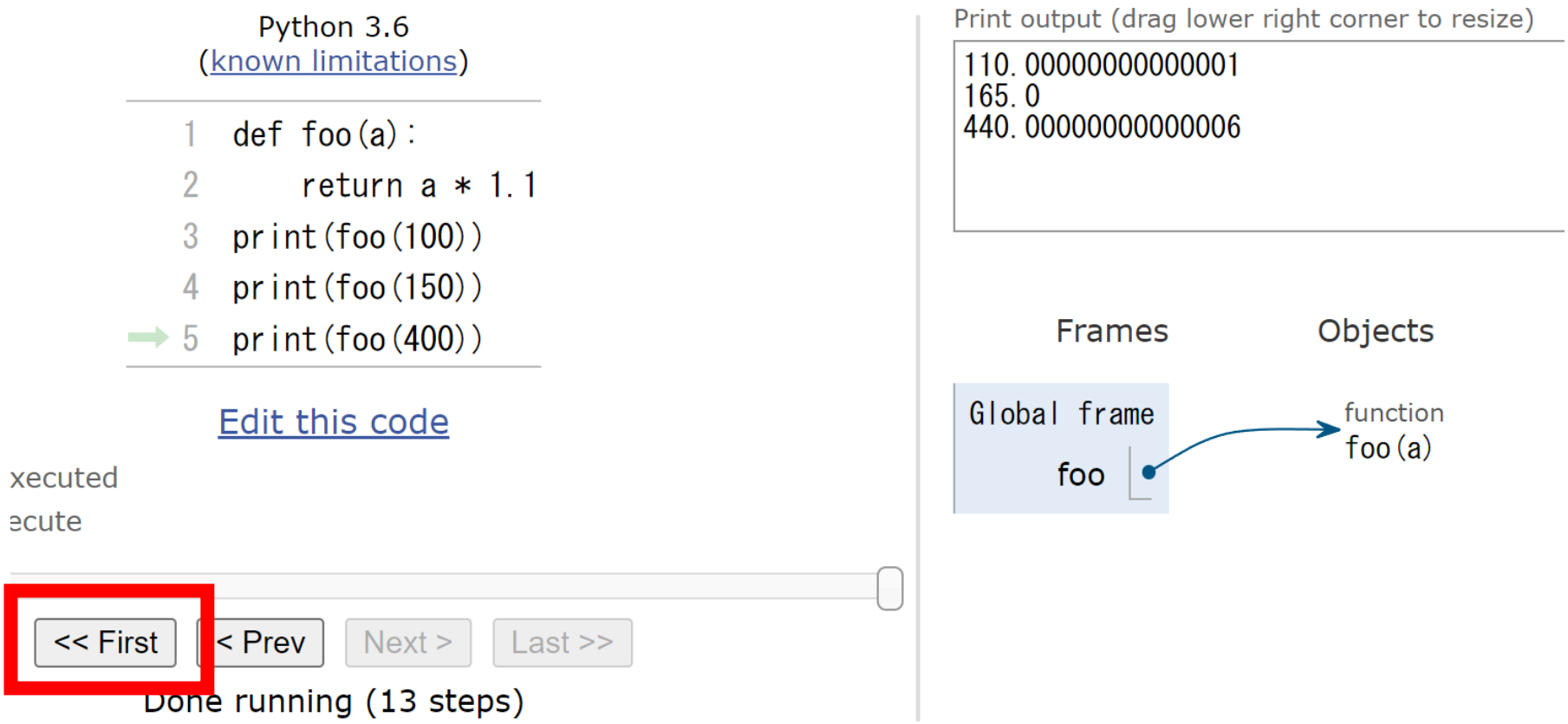
結果を確認

（計算誤差がある．
動作は正常）

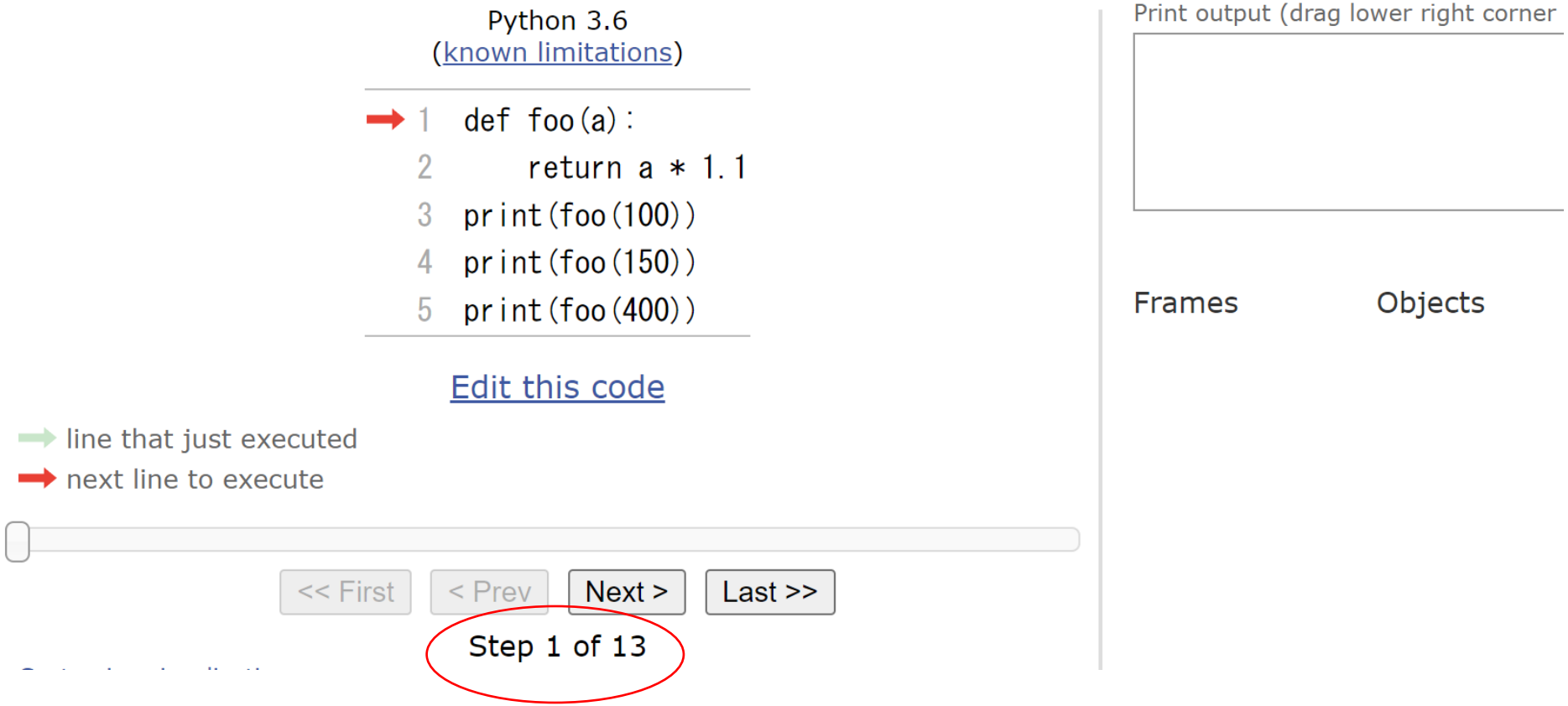
「return a * 1.1」の行は字下げが必要

「**Visual Execution**」をクリック．そして「**Last**」をクリック．結果を確認．
「**Edit this code**」をクリックすると，エディタの画面に戻る

②「First」をクリックして，最初に戻る

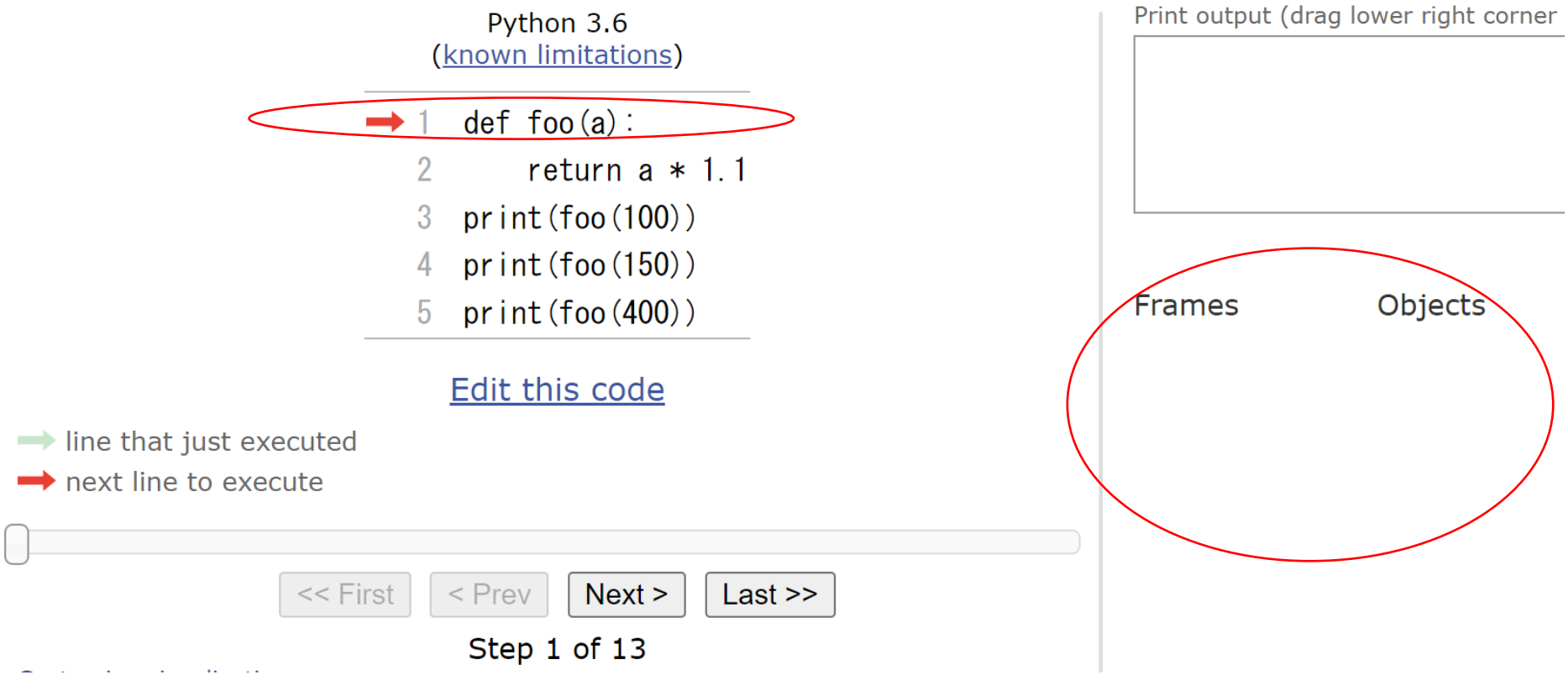


③「**Step 1 of 13**」と表示されているので，全部で，ステップ数は 13 あることが分かる（ステップ数と，プログラムの行数は違うもの）



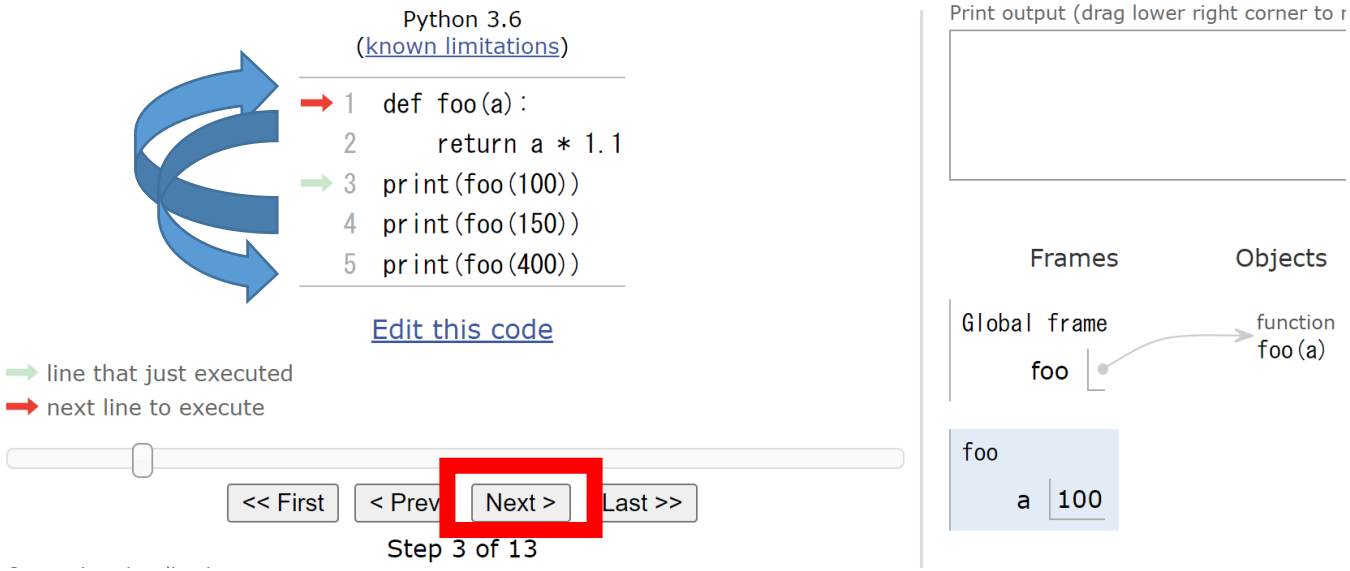
④ 最初に戻したので

- すべての**オブジェクト**は**消えている**
- **赤い矢印**は**先頭**のところに**戻っている**



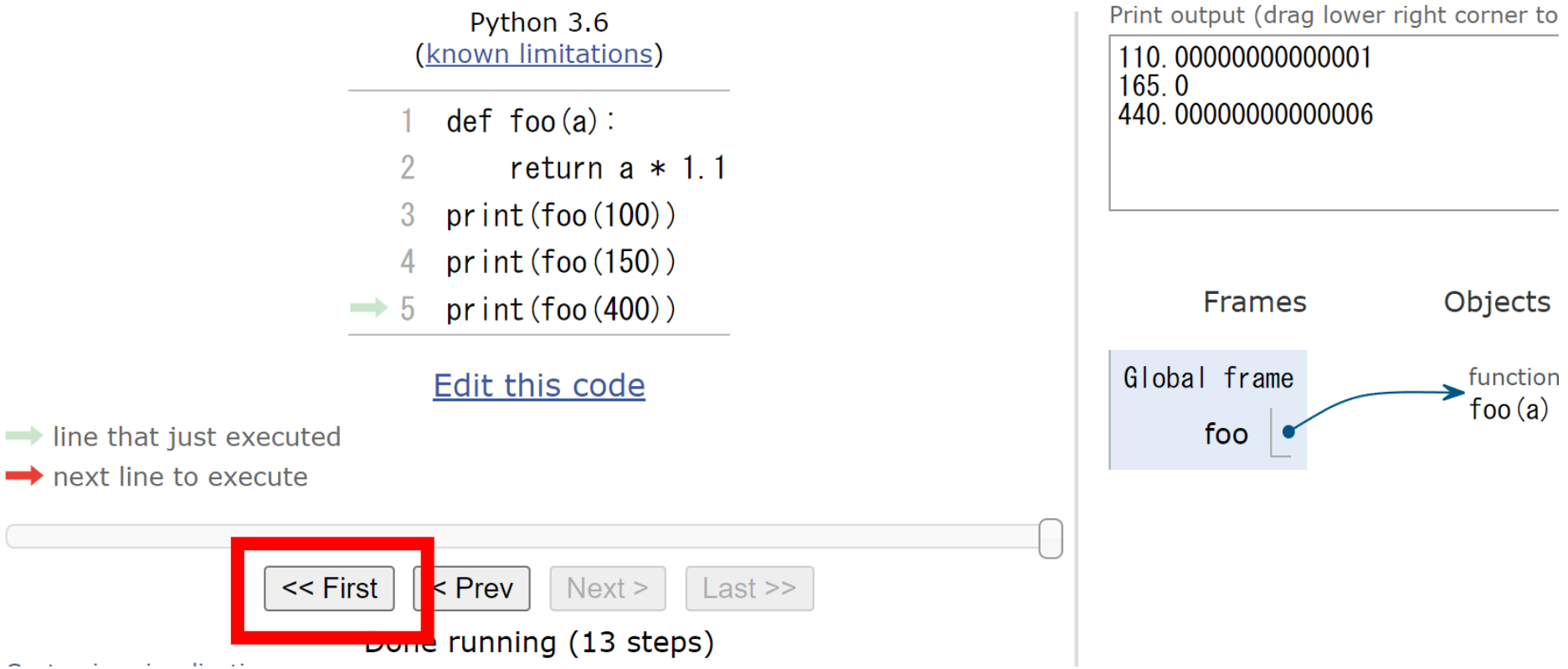
⑤ **ステップ実行**したいので，「**Next**」をクリックしながら，**矢印の動きを確認.**

※「**Next**」ボタンを何度か押し，それ以上進めなくなったら終了



見どころ
foo との間で
ジャンプする**ところ**

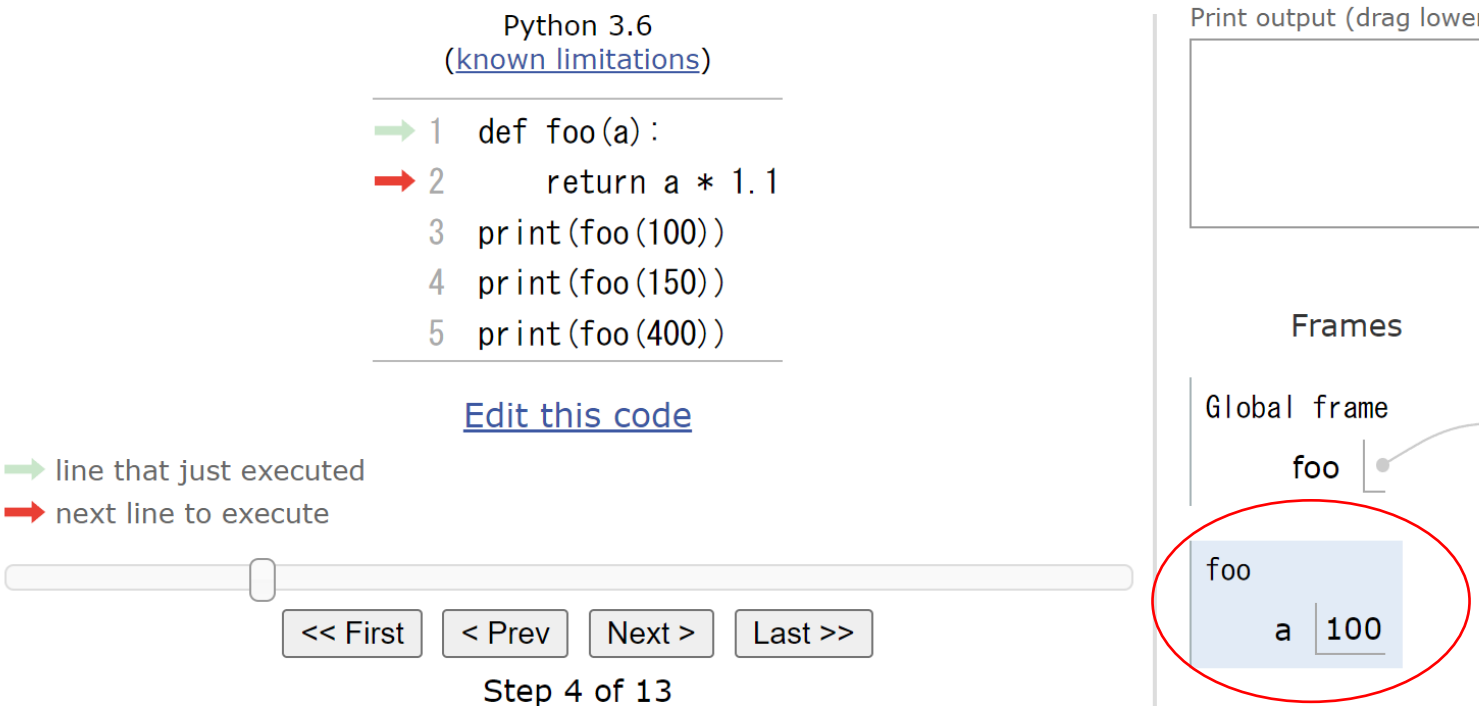
⑥ 終わったら，もう一度，「First」をクリックして，最初に戻る



⑦ もう一度，**ステップ実行**.

今度は、**緑の矢印**を見ながら、**変数 a** が**生成**、**消去**されるタイミングを確認

緑の矢印：**いま実行が終わった行**



関数 foo の実行中は，
変数 a が現れる

全体まとめ

```
1  print(100 * 1.1)
2  print(150 * 1.1)
3  print(200 * 1.1)
```

プログラム

Print output (drag lower right corner to resize)

```
110.00000000000001
165.0
220.00000000000003
```

Frames Objects

実行結果

```
1  def foo(a):
2      return a * 1.1
3  print(foo(100))
4  print(foo(150))
5  print(foo(400))
```

Print output (drag lower rigl

```
110.00000000000001
165.0
440.00000000000006
```

実行結果

３つの式を１つにまとめる

式「**a * 1.1**」を含む
関数 foo を**定義**

110.00000000000001
440.00000000000006
は計算誤差を含む
（誤作動やミスではない）

Python 関連ページ

- Python まとめページ

https://www.kkaneko.jp/tools/man/python.html

- Python 入門（スライド資料とプログラム例）

https://www.kkaneko.jp/pro/pf/index.html

- Python プログラミングの基本（スライド資料とプログラム例）

https://www.kkaneko.jp/pro/po/index.html

- Python プログラム例

https://www.kkaneko.jp/pro/python/index.html

- 人工知能の実行（Google Colaboratory を使用）

https://www.kkaneko.jp/ai/ni/index.html

- 人工知能の実行（Python を使用）（Windows 上）

https://www.kkaneko.jp/ai/deepim/index.html