Uncovering Python's surprises: a deep dive into gotchas

Mia Bajić

About me

- software engineer working at Ataccama focused on creating data products
- based in Prague, Czech Republic
- I like programming riddles and I like learning how Python works under the hood
- co-organizer of Prague Python meetups & co-organizer of PyCon CZ 2023

About the talk

- gotcha is a valid construct in a system, program or programming language that works as documented but is counter-intuitive
- examples I personally encountered when learning Python that surprised me and behaved differently than I expected
- common pitfalls to be avoided

Function arguments

Function arguments

```
>>> def double_int(number):
...     return 2 * number
```

Function arguments

```
>>> def double_int(number):
...   return 2 * number
>>> number = 5
>>> double_int(number)
10
```

Function arguments

```
>>> def double_int(number):
...   return 2 * number
>>> number = 5
>>> double_int(number)
10
>>> double_int(number)
10
```

Function arguments

```
>>> def double_list_ints(numbers):
...   for i in range(len(numbers)):
...     numbers[i] *= 2
...   return numbers
```

Function arguments

```python
>>> def double_list_ints(numbers):
...   for i in range(len(numbers)):
...     numbers[i] *= 2
...   return numbers
>>> numbers = [1, 2, 3]
>>> double_list_ints(numbers)
[2, 4, 6]
```

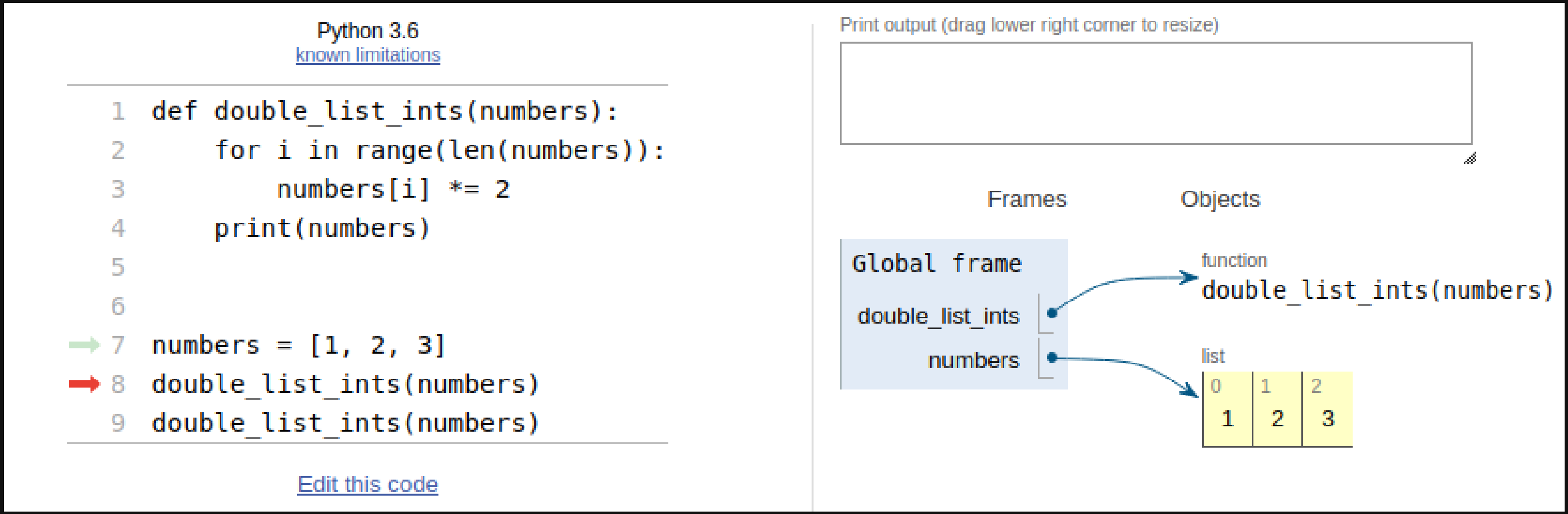
Function arguments

```
>>> def double_list_ints(numbers):
...   for i in range(len(numbers)):
...     numbers[i] *= 2
...   return numbers
>>> numbers = [1, 2, 3]
>>> double_list_ints(numbers)
[2, 4, 6]
>>> double_list_ints(numbers)
[4, 8, 12]
```

Function arguments

- in Python lists are mutable
- objects are not being copied when they're passed to the function - the original list was passed to the first function, it was modified by it and then it was passed to the second function
- any modification that is made to the object is visible outside the function

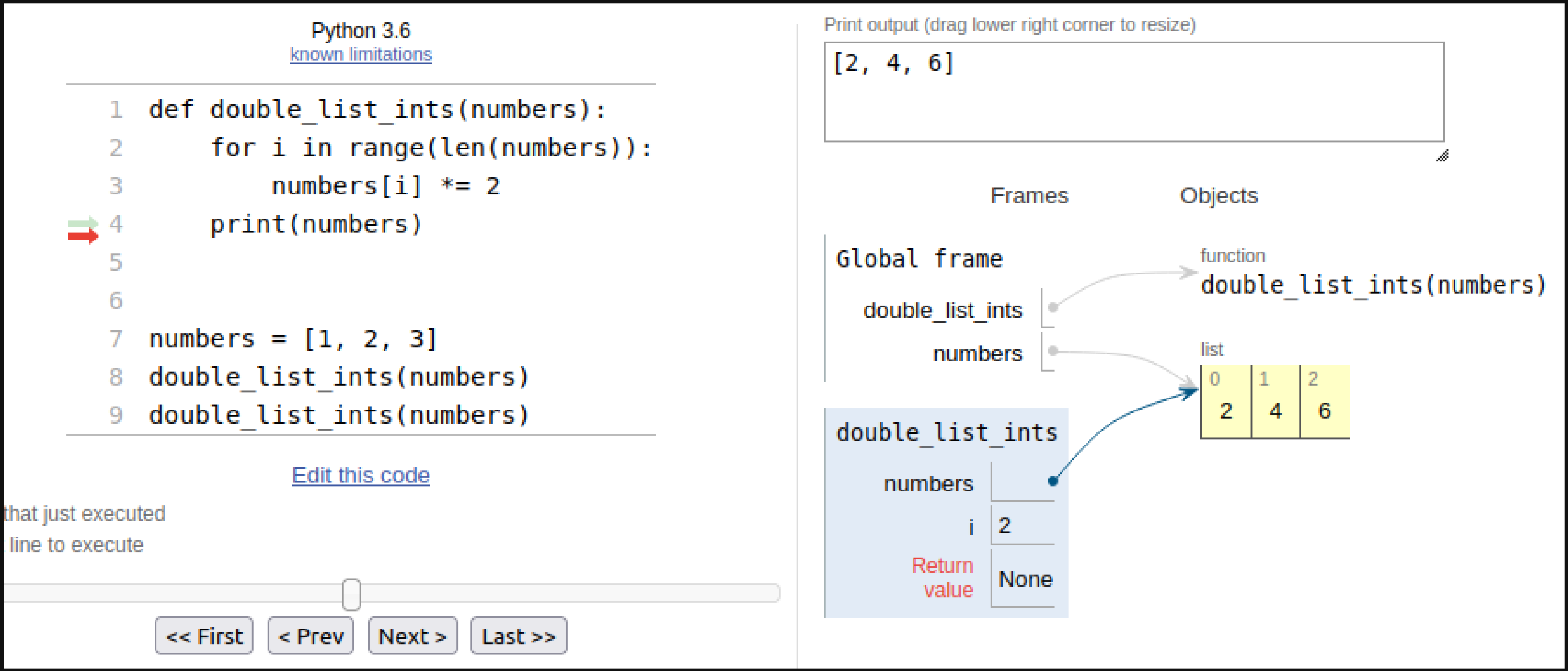
Function arguments



```
def double_list_ints(numbers):
    for i in range(len(numbers)):
        numbers[i] *= 2
    print(numbers)


numbers = [1, 2, 3]
double_list_ints(numbers)
double_list_ints(numbers)
```

Function arguments

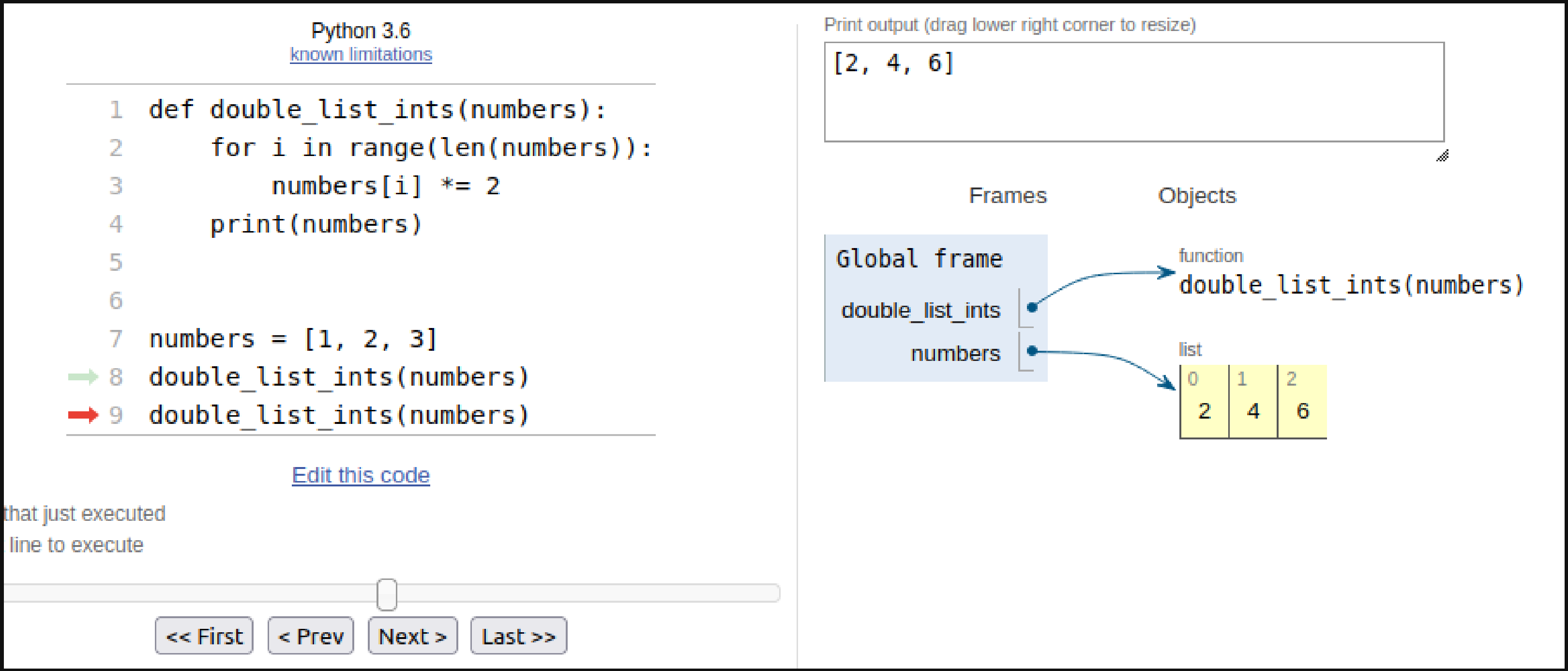


```
def double_list_ints(numbers):
    for i in range(len(numbers)):
        numbers[i] *= 2
    print(numbers)


numbers = [1, 2, 3]
double_list_ints(numbers)
double_list_ints(numbers)
```

```
[2, 4, 6]
```

Function arguments



Python 3.6
known limitations

```
def double_list_ints(numbers):
    for i in range(len(numbers)):
        numbers[i] *= 2
    print(numbers)


numbers = [1, 2, 3]
double_list_ints(numbers)
double_list_ints(numbers)
```

Edit this code

that just executed
line to execute

Print output (drag lower right corner to resize)

```
[2, 4, 6]
```

Function arguments

```
>>> def double(number=42):
```

Function arguments

```
>>> def double(number=42):
...     return 2 * number
```

Function arguments

```
>>> def double_integer(number=42):
...   return 2 * number
...
>>> double_integer()
84
```

Function arguments

```
>>> def double_integer(number=42):
...   return 2 * number
...
>>> double_integer()
84
>>> double_integer()
84
```

Function arguments

```
>>> def double_list_ints(numbers=[0, 1, 2, 3]):
```

Function arguments

```
>>> def double_list_ints(numbers=[0, 1, 2, 3]):
...     for i in range(len(numbers)):
```

Function arguments

```
>>> def double_list_ints(numbers=[0, 1, 2, 3]):
...   for i in range(len(numbers)):
...     numbers[i] *= 2
```

Function arguments

```
>>> def double_list_ints(numbers=[0, 1, 2, 3]):
...   for i in range(len(numbers)):
...     numbers[i] *= 2
...   return numbers
```

Function arguments

```
>>> def double_list_ints(numbers=[0, 1, 2, 3]):
...   for i in range(len(numbers)):
...     numbers[i] *= 2
...   return numbers
...
>>> double_list_ints()
[0, 2, 4, 6]
```

Function arguments

```
>>> def double_list_ints(numbers=[0, 1, 2, 3]):
...   for i in range(len(numbers)):
...     numbers[i] *= 2
...   return numbers
...
>>> double_list_ints()
[0, 2, 4, 6]
>>> double_list_ints()
[0, 4, 8, 12]
```

Function arguments

```
>>> def double_list_ints(numbers=[0, 1, 2, 3]):
...     for i in range(len(numbers)):
...         numbers[i] *= 2
...     return numbers
...
>>> double_list_ints()
[0, 2, 4, 6]
>>> double_list_ints()
[0, 4, 8, 12]
>>> double_list_ints()
[0, 8, 16, 24]
```

Function arguments

- default arguments are being evaluated only on its definition
- if mutable argument is passed, it will be modified by each function run

Function arguments

```
>>> def double_list_ints(numbers=None):
...   if numbers is None:
...     numbers = [0, 1, 2, 3]
...   for i in range(len(numbers)):
...     numbers[i] *= 2
...   return numbers
...
>>> double_list_ints()
[0, 2, 4, 6]
>>> double_list_ints()
[0, 2, 4, 6]
>>> double_list_ints()
[0, 2, 4, 6]
```

Aliasing

Aliasing

```
>>> numbers = [1, 2, 3]
```

Aliasing

```
>>> numbers = [1, 2, 3]
>>> second_list = numbers
```

Aliasing

```
>>> numbers = [1, 2, 3]
>>> second_list = numbers
>>> numbers.append(4)
```

Aliasing

```
>>> numbers = [1, 2, 3]
>>> second_list = numbers
>>> numbers.append(4)
>>> numbers
[1, 2, 3, 4]
```

Aliasing

```
>>> numbers = [1, 2, 3]
>>> second_list = numbers
>>> numbers.append(4)
>>> numbers
[1, 2, 3, 4]
>>> second_list
[1, 2, 3, 4]
```

Aliasing

```
>>> numbers = [1, 2, 3]
>>> second_list = numbers
>>> numbers.append(4)
>>> numbers
[1, 2, 3, 4]
>>> second_list
[1, 2, 3, 4]
>>> numbers is second_list
True
```

Aliasing

```
>>> number = 1
```

Aliasing

```
>>> number = 1
>>> second_number = number
```

Aliasing

```
>>> number = 1
>>> second_number = number
>>> number is second_number
True
```

Aliasing

```
>>> number = 1
>>> second_number = number
>>> number is second_number
True
>>> number = 3
>>> number
3
```

Aliasing

```
>>> number = 1
>>> second_number = number
>>> number is second_number
True
>>> number = 3
>>> number
3
>>> second_number
1
```

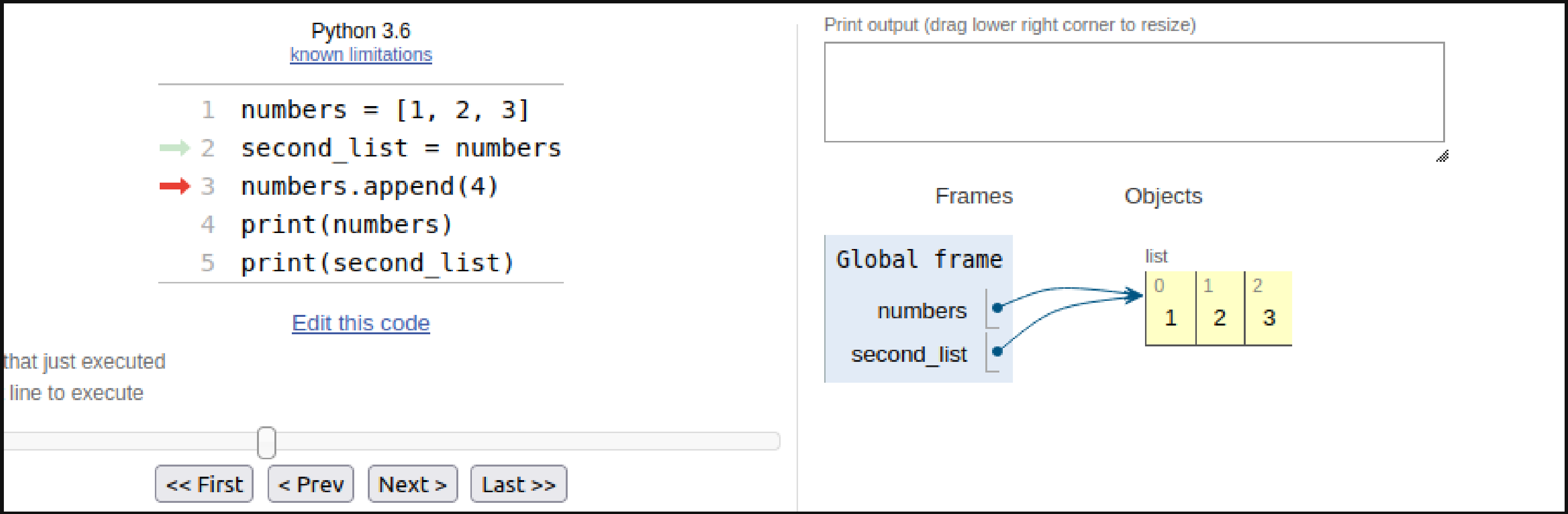
Aliasing

```
>>> number = 1
>>> second_number = number
>>> number is second_number
True
>>> number = 3
>>> number
3
>>> second_number
1
>>> number is second_number
False
```

Aliasing

- when value of mutable type is modified all names referring to the same object see the change
- values of immutable types cannot be modified
    - you need to make a new object and reassign the variable to reference the new object
    - other references are not being updated and they still hold the original value

Aliasing



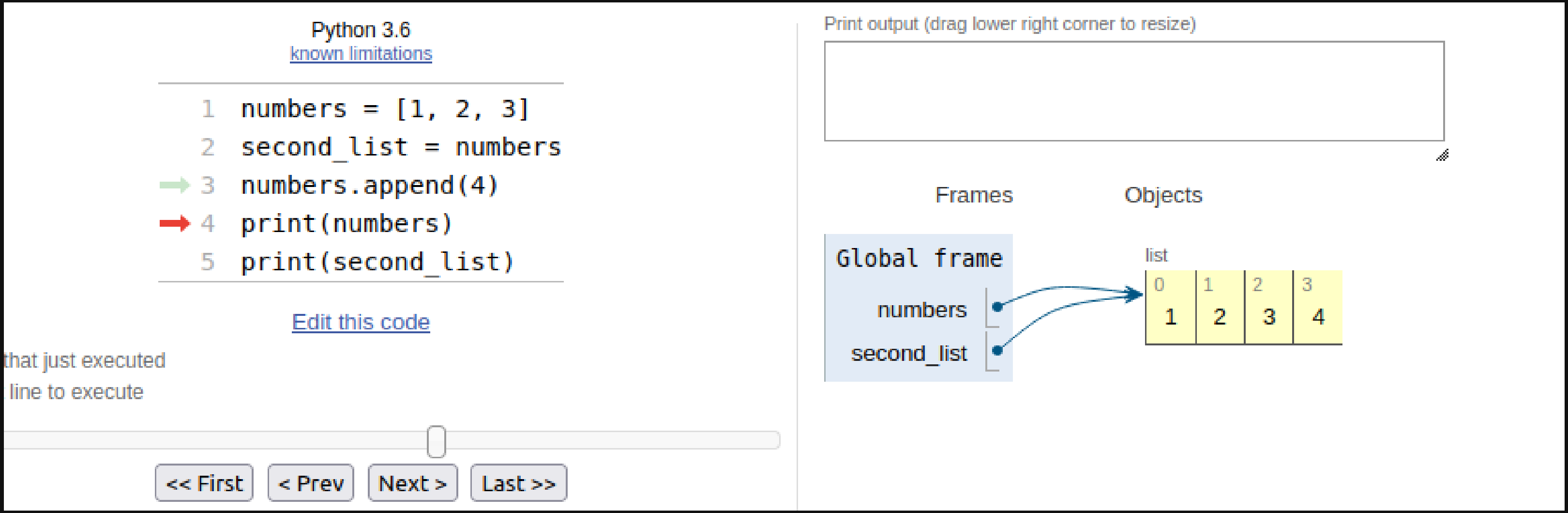
Python 3.6
known limitations

```
numbers = [1, 2, 3]
second_list = numbers
numbers.append(4)
print(numbers)
print(second_list)
```

Edit this code

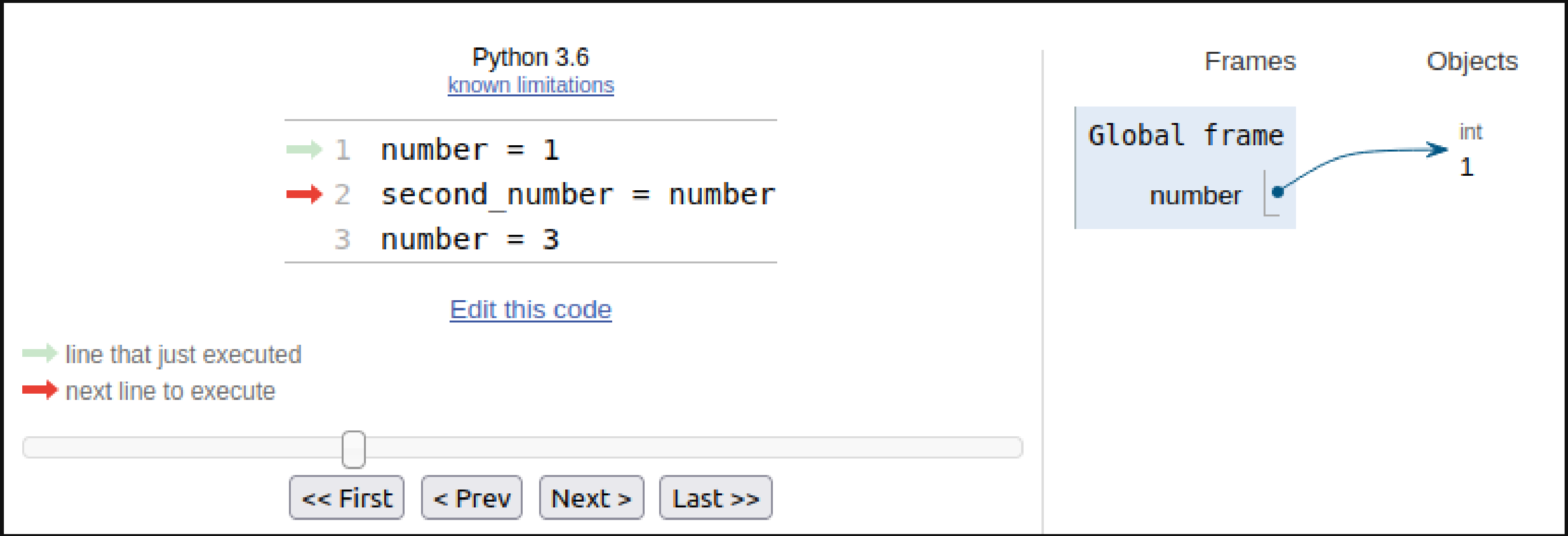
that just executed
line to execute

Aliasing



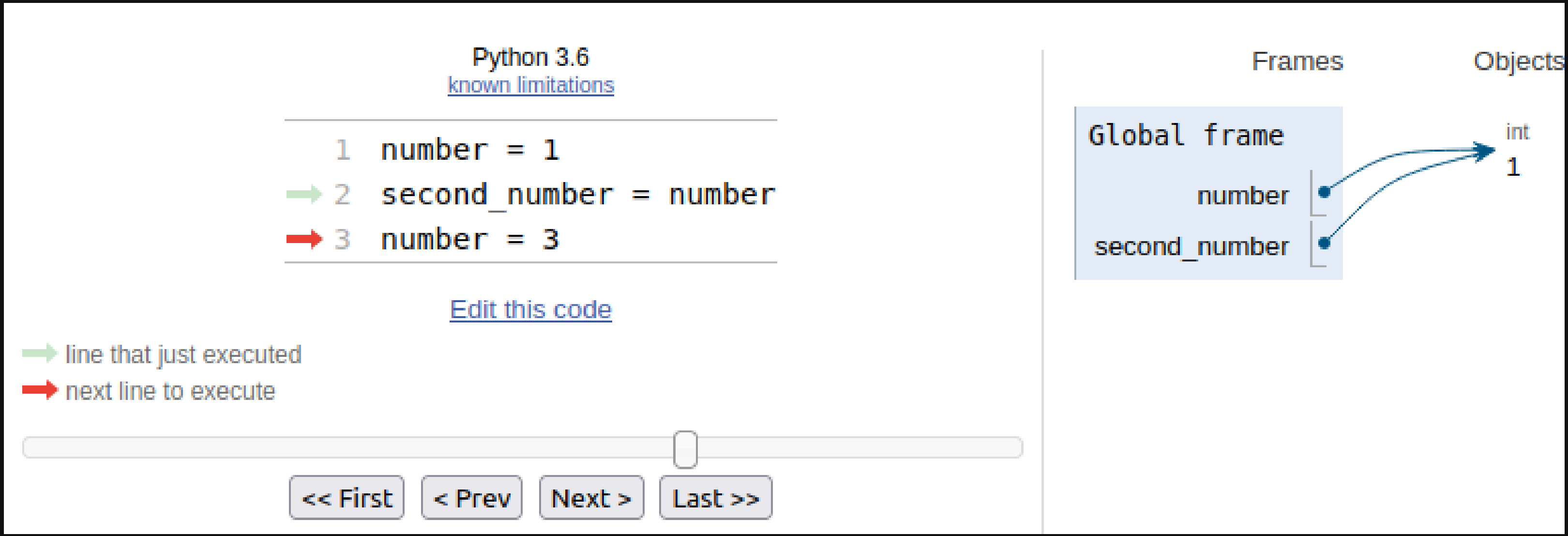
```
numbers = [1, 2, 3]
second_list = numbers
numbers.append(4)
print(numbers)
print(second_list)
```

Aliasing



```
number = 1
second_number = number
number = 3
```

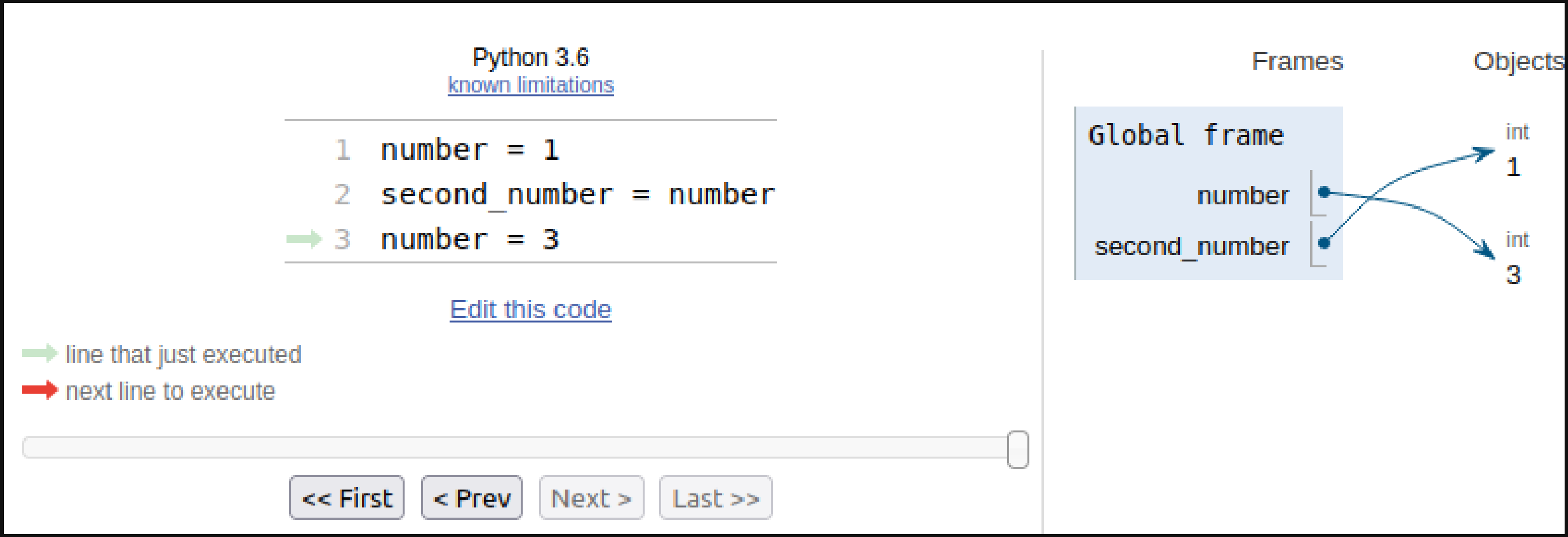
Aliasing


Python 3.6
known limitations

```
number = 1
second_number = number
number = 3
```

Edit this code

Aliasing



Python 3.6
known limitations

```
number = 1
second_number = number
number = 3
```

Edit this code

line that just executed
next line to execute

Interning

Interning

```
>>> a = "hi"
```

Interning

```
>>> a = "hi"
>>> b = "hi"
```

Interning

```
>>> a = "hi"
>>> b = "hi"
>>> a is b
True
```

Interning

```
>>> a = "hi"
>>> b = "hi"
>>> a is b
True
>>> a = "hi!"
```

Interning

```
>>> a = "hi"
>>> b = "hi"
>>> a is b
True
>>> a = "hi!"
>>> b = "hi!"
```

Interning

```
>>> a = "hi"
>>> b = "hi"
>>> a is b
True
>>> a = "hi!"
>>> b = "hi!"
>>> a is b
False
```

Interning

```
>>> a = "hi there"
>>> b = "hi there"
```

Interning

```
>>> a = "hi there"
>>> b = "hi there"
>>> a is b
False
```

Interning

- interning is a CPython optimization where multiple variables may reference the same string object
- the aim is to reuse immutable objects instead of creating new ones
- the decision whether the string is interned is implementation specific
- for example CPython automatically interns small strings and identifier names

Equivalence of numbers

Equivalence of numbers

```
>>> numbers = {1.5: "a", 1.0: "b", 1: "c"}
```

Equivalence of numbers

```
>>> numbers = {1.5: "a", 1.0: "b", 1: "c"}
>>> numbers[1.5]
'a'
```

Equivalence of numbers

```
>>> numbers = {1.5: "a", 1.0: "b", 1: "c"}
>>> numbers[1.5]
'a'
>>> numbers[1]
'c'
```

Equivalence of numbers

```
>>> numbers = {1.5: "a", 1.0: "b", 1: "c"}
>>> numbers[1.5]
'a'
>>> numbers[1]
'c'
>>> numbers[1.0]
'c'
```

Equivalence of numbers

- Python keys are unique by its equivalence
- values 1 and 1.0 are different objects and different types, but they are equal
- when adding another item to dictionary, if keys are equal then the original value is overwritten

Equivalence of numbers

- Python keys are unique by its equivalence
- values 1 and 1.0 are different objects and different types, but they are equal
- when adding another item to dictionary, if keys are equal then the original value is overwritten

```
>>> numbers = {1.5: "a", 1.0: "b", 1: "c"}
>>> numbers
{1.5: 'a', 1.0: 'c'}
```

List iteration

List iteration

```
>>> a = [0, 1, 2, 3]
```

List iteration

```
>>> a = [0, 1, 2, 3]
>>> for a[-1] in a:
```

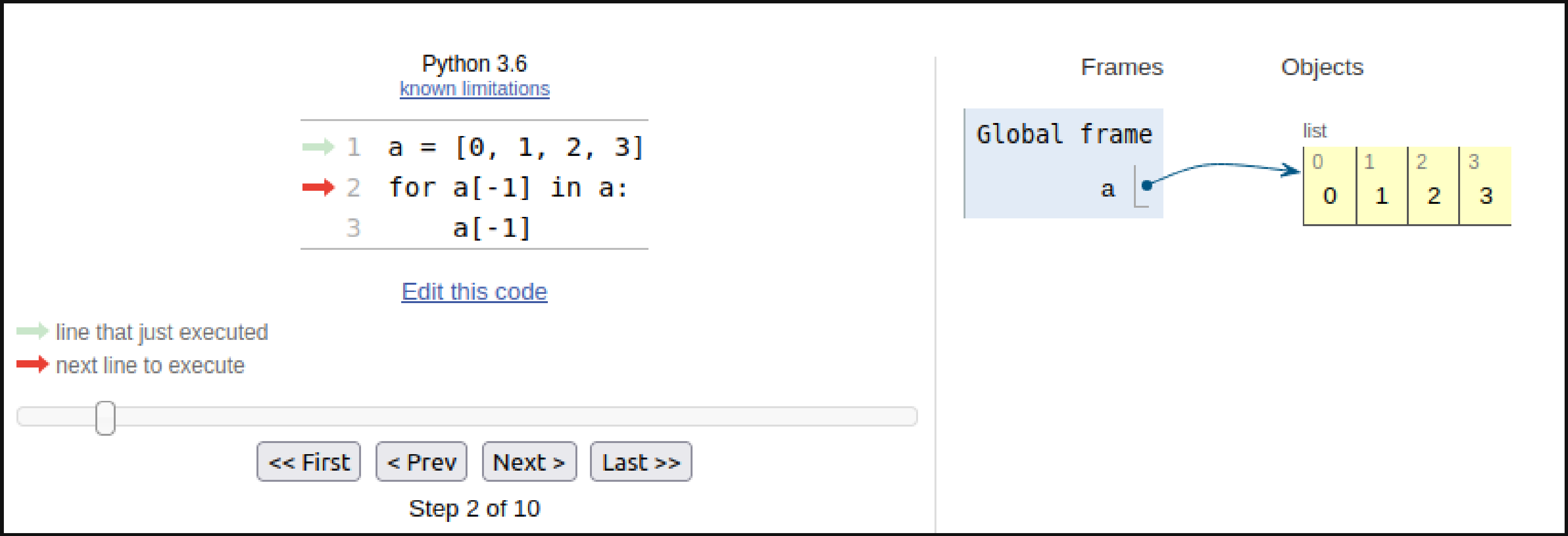
List iteration

```
>>> a = [0, 1, 2, 3]
>>> for a[-1] in a:
...     print(a[-1])
```

List iteration

```
>>> a = [0, 1, 2, 3]
>>> for a[-1] in a:
...     print(a[-1])
...
0
1
2
2
```

List iteration



Python 3.6
known limitations

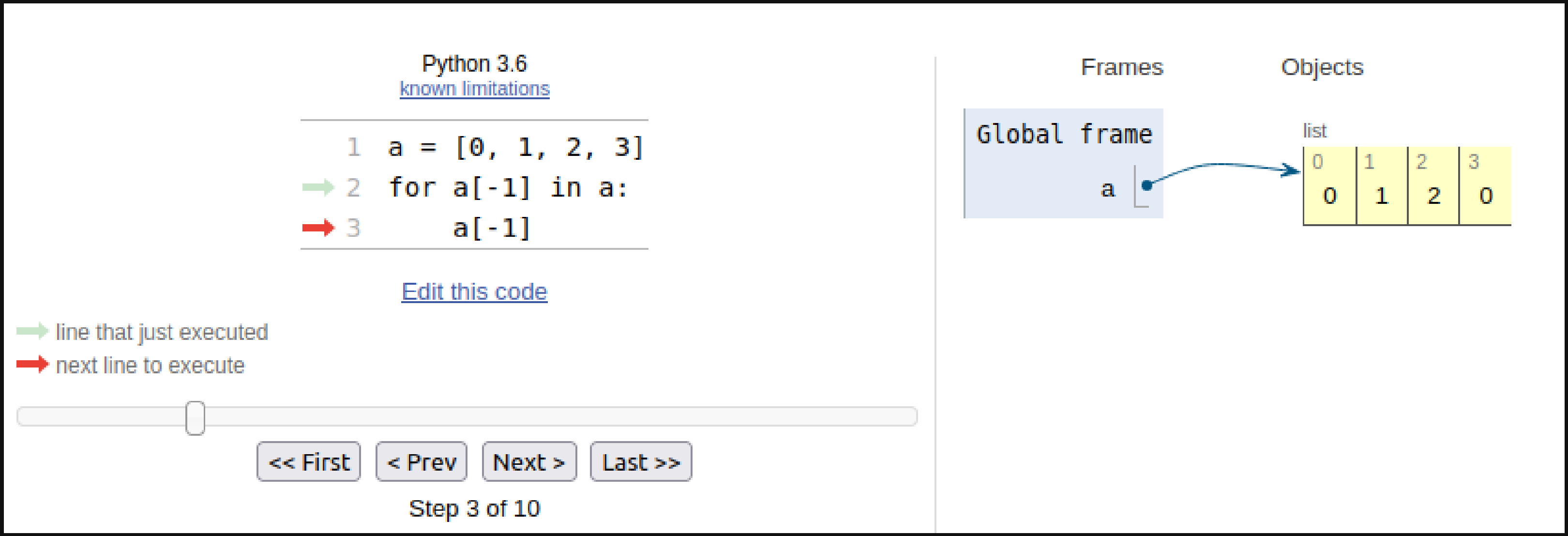
```
a = [0, 1, 2, 3]
for a[-1] in a:
    a[-1]
```

Edit this code

line that just executed
next line to execute

Step 2 of 10

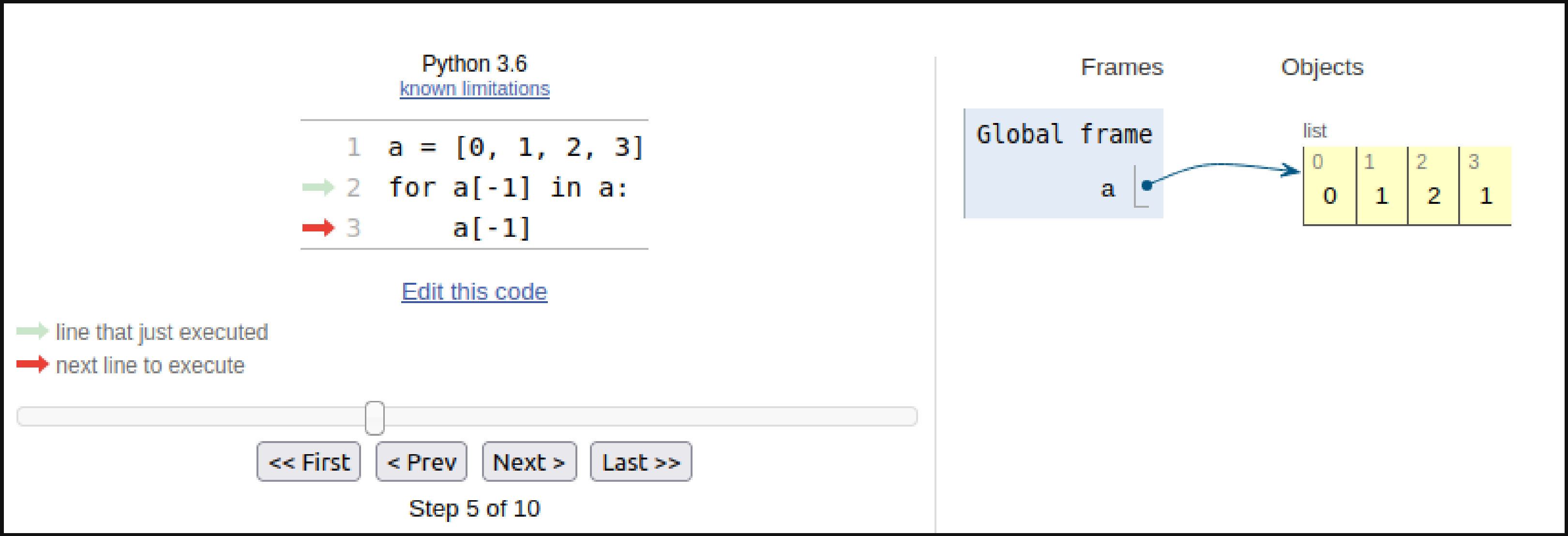
List iteration



Python 3.6
known limitations

```
a = [0, 1, 2, 3]
for a[-1] in a:
    a[-1]
```

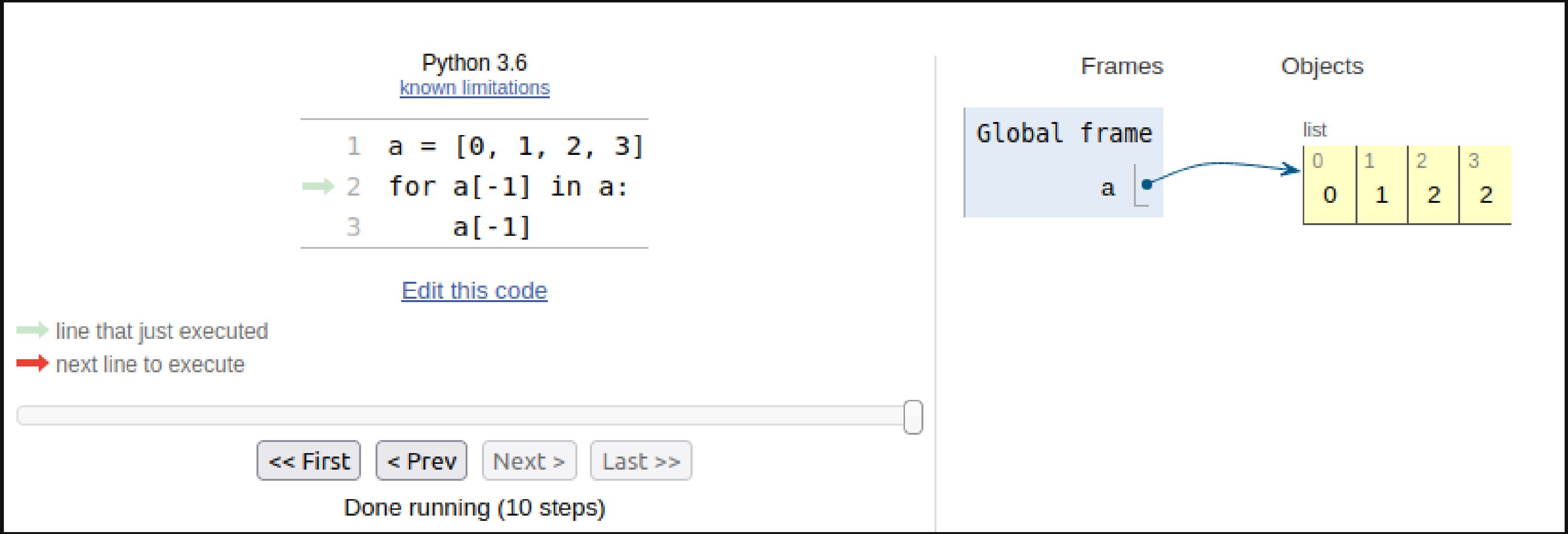
List iteration


Python 3.6
known limitations

```
a = [0, 1, 2, 3]
for a[-1] in a:
    a[-1]
```

Edit this code

List iteration


```
a = [0, 1, 2, 3]
for a[-1] in a:
    a[-1]
```

List iteration

- for a[-1] in a loops over list and temporary stores the value of the current element into a[-1]
- at the last iteration the last value stored is 2, therefore it's printed twice

Sources

- Anthony Shaw: CPython internals
- Ned Batchelder: Facts and Myths about Python names and values www.youtube.com/watch?v=_AEJHKhttpsGk9ns
- WTF Python: https://github.com/satwikkansal/wtfpython
- Aliasing: https://www.cs.cmu.edu/~15110-s20/slides/week6-2-aliasing.pdf
- https://docs.python.org/3/

Contact

- linkedin: /in/mia-bajic/
- gmail: miabajic.miabajic@gmail.com

Thank you for attention!