Module 14

Error Handling

Motivation

- Suppose we have this code:

```
result = input('Number: ')          # get number from user
x = float(result)                   # convert string to float
print('The next number is '+str(x+1))
```

- What if user mistypes?

```
Number: 12a
Traceback (most recent call last):
  File "prompt.py", line 13, in <module>
    x = float(result)
ValueError: could not convert string to float: '12a'
```

Ideally Would Handle with Conditional

```
result = input('Number: ')      # get number from user
if is_float(result):

    x = float(input)            # convert to float
    print('The next number is '+str(x+1))
else:
    print('That is not a number!')
```

Does not Exist

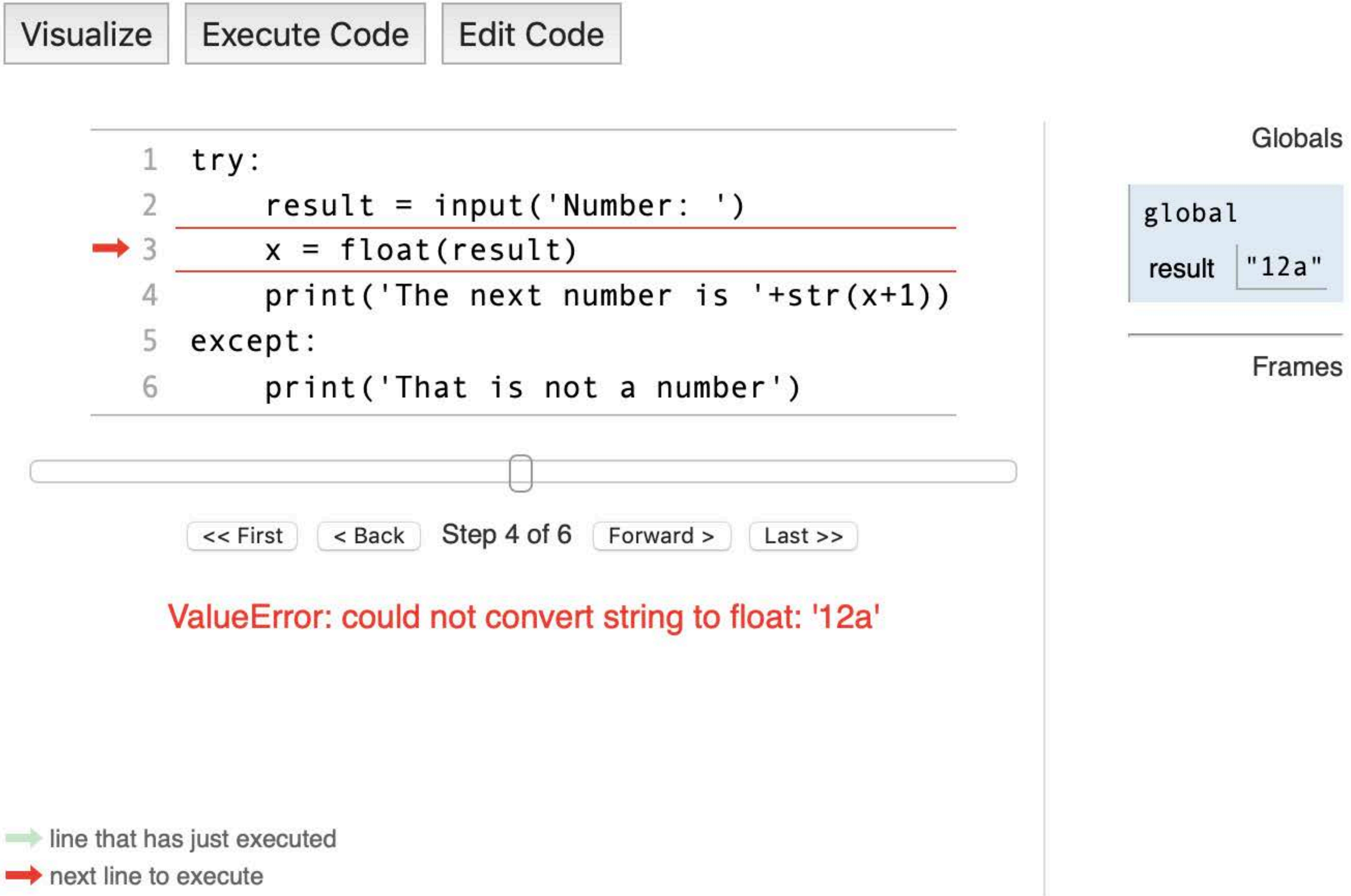
Using Try-Except

```
try:
    result = input('Number: ')      # get number
    x = float(input)                # convert to float
    print('The next number is '+str(x+1))
except:
    print('That is not a number!')
```

Similar to if-else

- But always does the try block
- Might not do **all** of the try block

Python Tutor Example



```
try:
    result = input('Number: ')
    x = float(result)
    print('The next number is '+str(x+1))
except:
    print('That is not a number')
```

ValueError: could not convert string to float: '12a'

global

result	"12a"

➡ line that has just executed

➡ next line to execute

A Problematic Function

```
def is_number(s):
    """Returns: True if string s can be cast to a float

    Examples:  is_number('a') is False
               is_number('12') is True
               is_number('12.5') is True
               is_number('1e-2') is True
               is_number(‘0-1') is False

    Precondition: s is a string"""
```

These examples seem a bit overwhelming

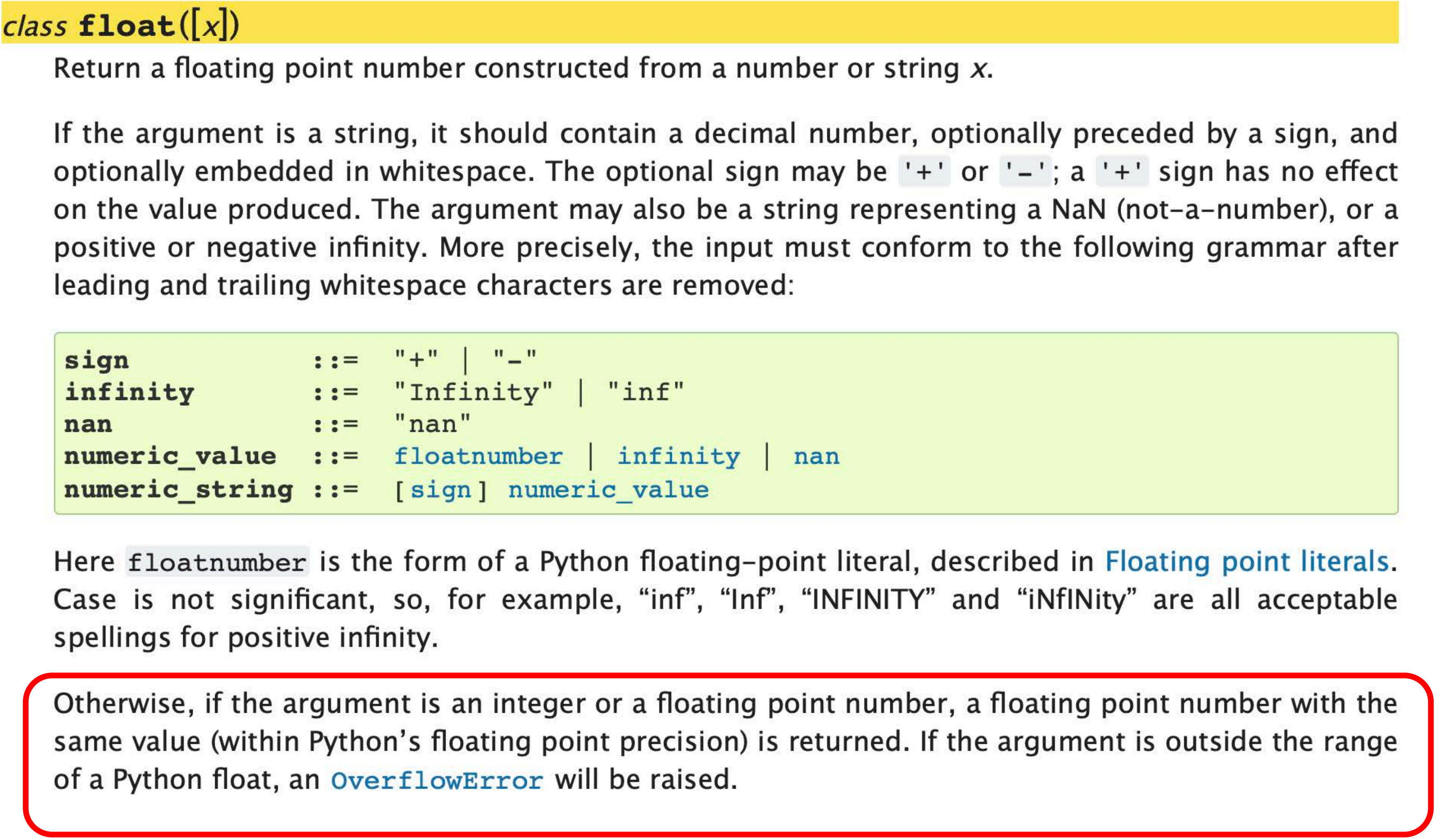
A Problematic Function

```
def is_number(s):
    """Returns: True if string s can be cast to a float

    Precondition: s is a string"""
```

- Complications (It is a mess)
  - Everything must be digit, e, minus, or period
  - Period can only happen once
  - Minus can only happen after e
  - The e can only be second

An Observation



class **float**([*x*])

Return a floating point number constructed from a number or string *x*.

If the argument is a string, it should contain a decimal number, optionally preceded by a sign, and optionally embedded in whitespace. The optional sign may be `'+'` or `'-'`; a `'+'` sign has no effect on the value produced. The argument may also be a string representing a NaN (not-a-number), or a positive or negative infinity. More precisely, the input must conform to the following grammar after leading and trailing whitespace characters are removed:

```
sign           ::=  "+" | "-"
infinity       ::=  "Infinity" | "inf"
nan            ::=  "nan"
numeric_value  ::=  floatnumber | infinity | nan
numeric_string ::=  [sign] numeric_value
```

Here `floatnumber` is the form of a Python floating-point literal, described in Floating point literals. Case is not significant, so, for example, “inf”, “Inf”, “INFINITY” and “iNfINity” are all acceptable spellings for positive infinity.

Otherwise, if the argument is an integer or a floating point number, a floating point number with the same value (within Python’s floating point precision) is returned. If the argument is outside the range of a Python float, an `OverflowError` will be raised.

Taking Advantage of Errors

```
def is_float(s):
    """Returns: True if string s can be cast to a float

    Precondition: s is a string"""
    try:
        x = float(s)
        return True
    except:
        return False
```

Conversion to a float might fail

If attempt succeeds, string s is a float

Otherwise, it is not

A Design Philosophy Difference

- Conditionals are **asking for permission**
  - Check if a property holds
  - The body proceeds if it is safe
- Try-Except is **asking for forgiveness**
  - Assumes that a property always holds
  - Recovers if it does not
- Python often prefers the **latter**
  - But this is largely unique to Python
  - Only because errors are “relatively” cheap

A Design Philosophy Difference

- Conditionals are **asking for permission**
  - Check if a property holds
  - The body proceeds if it is safe
- Try-Except is **asking for f**
  - Assumes t
- P ers the **latter**
  - ut this is largely unique to Python
  - Only because errors are “relatively” cheap

But still use try-except sparingly.
Only when it simplifies code a lot.

Errors and the Call Stack

```
#

d
    return function_2(x,y)

def function_2(x,y):
    return function_3(x,y)

def function_3(x,y):
    return x/y # crash here

if
```

Script code. Global space

Where error occurred (or where was found)

Crashes produce the call stack:

```
Traceback (most recent call last):
  File "error.py", line 20, in <module>
    print(function_1(1,0))
  File "error.py", line 8, in function_1
    return function_2(x,y)
  File "error.py", line 12, in function_2
    return function_3(x,y)
  File "error.py", line 16, in function_3
    return x/y
```

Make sure you can see line numbers in Atom.

Try-Except and the Call Stack

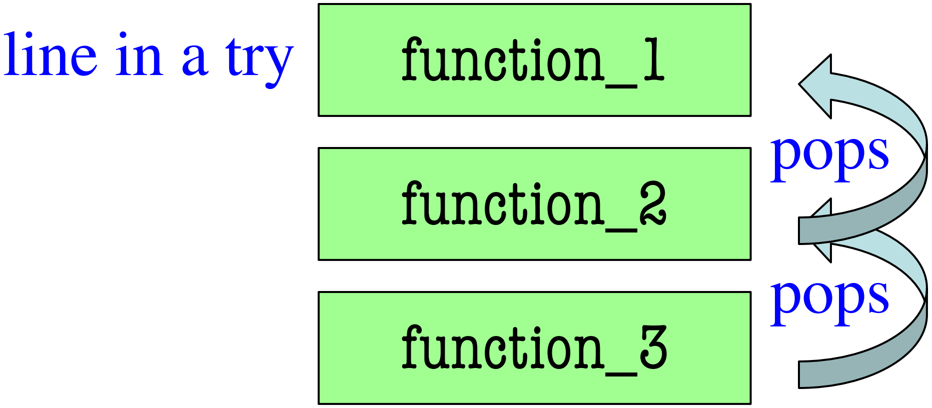
```
# recover.py

def function_1(x,y):
    try:
        return function_2(x,y)
    except:
        return float('inf')

def function_2(x,y):
    return function_3(x,y)

def function_3(x,y):
    return x/y # crash here
```

- Error "pops" frames off stack
  - Starts from the stack bottom
  - Continues until it sees that current line is in a try-block
  - Jumps to except, and then proceeds as if no error



Tracing Control Flow

```
def first(x):
    print('Starting first.')
    try:
        second(x)
    except:
        print('Caught at first')
    print('Ending first')


def second(x):
    print('Starting second.’)
    third(x)
    print('Ending second')
```

```
def third(x):
    print('Starting third.')
    assert x < 1
    print('Ending third.')
```

What is the output of first(2)?

Tracing Control Flow

```
def first(x):
    print('Starting first.')
    try:
        second(x)
    except:
        print('Caught at first')
    print('Ending first')

def second(x):
    print('Starting second.’)
    third(x)
    print('Ending second')
```

```
def third(x):
    print('Starting third.')
    assert x < 1
    print('Ending third.')
```

What is the output of first(2)?

```
'Starting first.'
'Starting second.'
'Starting third.'
'Caught at first'
'Ending first'
```

Tracing Control Flow

```
def first(x):
    print('Starting first.')
    try:
        second(x)
    except:
        print('Caught at first')
    print('Ending first')

def second(x):
    print('Starting second.')
    try:
        third(x)
    except:
        print('Caught at second')
    print('Ending second')
```

```
def third(x):
    print('Starting third.')
    assert x < 1
    print('Ending third.')
```

What is the output of first(2)?

Tracing Control Flow

```
def first(x):
    print('Starting first.')
    try:
        second(x)
    except:
        print('Caught at first')
    print('Ending first')

def second(x):
    print('Starting second.')
    try:
        third(x)
    except:
        print('Caught at second')
    print('Ending second')
```

```
def third(x):
    print('Starting third.')
    assert x < 1
    print('Ending third.')
```

What is the output of first(2)?

```
'Starting first.'
'Starting second.'
'Starting third.'
'Caught at second'
'Ending second'
'Ending first'
```

Tracing Control Flow

```
def first(x):
    print('Starting first.')
    try:
        second(x)
    except:
        print('Caught at first')
    print('Ending first')


def second(x):
    print('Starting second.')
    try:
        third(x)
    except:
        print('Caught at second')
    print('Ending second')
```

```
def third(x):
    print('Starting third.')
    assert x < 1
    print('Ending third.')
```

What is the output of first(0)?

Tracing Control Flow

```
def first(x):
    print('Starting first.')
    try:
        second(x)
    except:
        print('Caught at first')
    print('Ending first')

def second(x):
    print('Starting second.')
    try:
        third(x)
    except:
        print('Caught at second')
    print('Ending second')
```

```
def third(x):
    print('Starting third.')
    assert x < 1
    print('Ending third.')
```

What is the output of first(0)?

```
'Starting first.'
'Starting second.'
'Starting third.'
'Ending third'
'Ending second'
'Ending first'
```

Testing: Code Coverage

- Remember testing for `if-elif-else`
  - Needed a test for each possible branch
  - We called this **code coverage**
- Need a similar approach for `try-except`
  - Need a test for the try and the except
  - But harder to identify except; no guards
  - Have to identify all the ways can crash
  - Requires viewing code line by line

An Example

```
def eval_frac(s):
    """Returns: string s evaluated as a fraction (or None)
    Precondition: s is a string"""
    try:
        pos = s.find('/')
        top = int(s[:pos])                      # Error?
        bot = int(s[pos+1:])                    # Error?
        return top//bot                         # Error?
    except:
        return None
```

See test script