61A Lecture 25

Announcements

Pairs Review

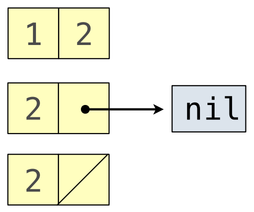
Pairs and Lists

In the late 1950s, computer scientists used confusing names

- **cons:** Two-argument procedure that creates a pair
- **car:** Procedure that returns the first element of a pair
- **cdr:** Procedure that returns the second element of a pair
- **nil:** The empty list
- A (non-empty) list in Scheme is a pair in which the second element is **nil** or a Scheme list
- **Important!** Scheme lists are written in parentheses separated by spaces
- A dotted list has some value for the second element of the last pair that is not a list

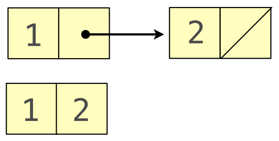
(cons 1 2)

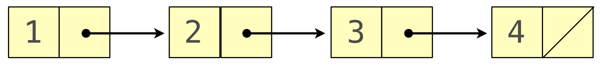
(cons 2 nil)



```
> (cons 1 (cons 2 nil))
(1 2)
> (define x (cons 1 2))
> x
(1 . 2)
> (car x)
1
> (cdr x)
2
> (cons 1 (cons 2 (cons 3 (cons 4 nil))))
(1 2 3 4)
```

Not a well-formed list!





(Demo)

Exceptions

Today's Topic: Handling Errors

Sometimes, computer programs behave in non-standard ways

- A function receives an argument value of an improper type
- Some resource (such as a file) is not available
- A network connection is lost in the middle of data transmission

```
1100  Started Cosine Tape (Sine check)
1525  Started Mult + Adder Test.
1545  Relay #70 Panel F
      (moth) in relay.
      First actual case of bug being found.
1630  antangent started.
1700  closed down.
```

Grace Hopper's Notebook, 1947, Moth found in a Mark II Computer

Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions

Python raises an exception whenever an error occurs

Exceptions can be handled by the program, preventing the interpreter from halting

Unhandled exceptions will cause Python to halt execution and print a stack trace

Mastering exceptions:

Exceptions are objects! They have classes with constructors.

They enable non-local continuations of control

If **f** calls **g** and **g** calls **h**, exceptions can shift control from **h** to **f** without waiting for **g** to return.

(Exception handling tends to be slow.)

Raising Exceptions

Assert Statements

Assert statements raise an exception of type AssertionError

```
assert <expression>, <string>
```

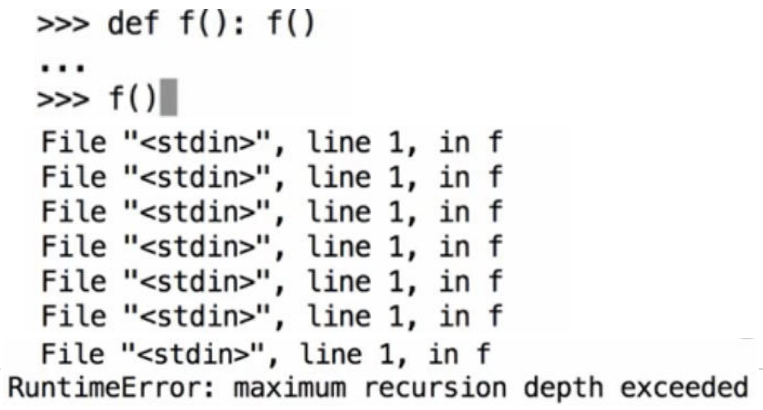
Assertions are designed to be used liberally. They can be ignored to increase efficiency by running Python with the "-O" flag; "O" stands for optimized

```
python3 -O
```

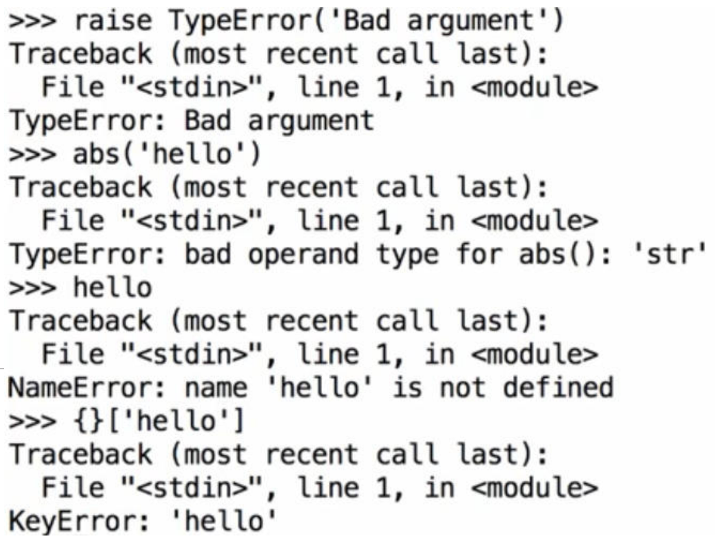
Whether assertions are enabled is governed by a bool __debug__

(Demo)

Raise Statements



```
>>> def f(): f()
...
>>> f()
File "<stdin>", line 1, in f
File "<stdin>", line 1, in f
File "<stdin>", line 1, in f
File "<stdin>", line 1, in f
File "<stdin>", line 1, in f
File "<stdin>", line 1, in f
File "<stdin>", line 1, in f
RuntimeError: maximum recursion depth exceeded
```



```
>>> raise TypeError('Bad argument')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Bad argument
>>> abs('hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'
>>> hello
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'hello' is not defined
>>> {}['hello']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'hello'
```

Exceptions are raised with a raise statement

raise <expression>

<expression> must evaluate to a subclass of BaseException or an instance of one

Exceptions are constructed like any other object. E.g., TypeError('Bad argument!')

TypeError -- A function was passed the wrong number/type of argument

NameError -- A name wasn't found

KeyError -- A key wasn't found in a dictionary

RuntimeError -- Catch-all for troubles during interpretation

(Demo)

Try Statements

Try Statements

Try statements handle exceptions

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

Execution rule:

The <try suite> is executed first

If, during the course of executing the <try suite>,
an exception is raised that is not handled otherwise, and

If the class of the exception inherits from <exception class>, then

The <except suite> is executed, with <name> bound to the exception

Handling Exceptions

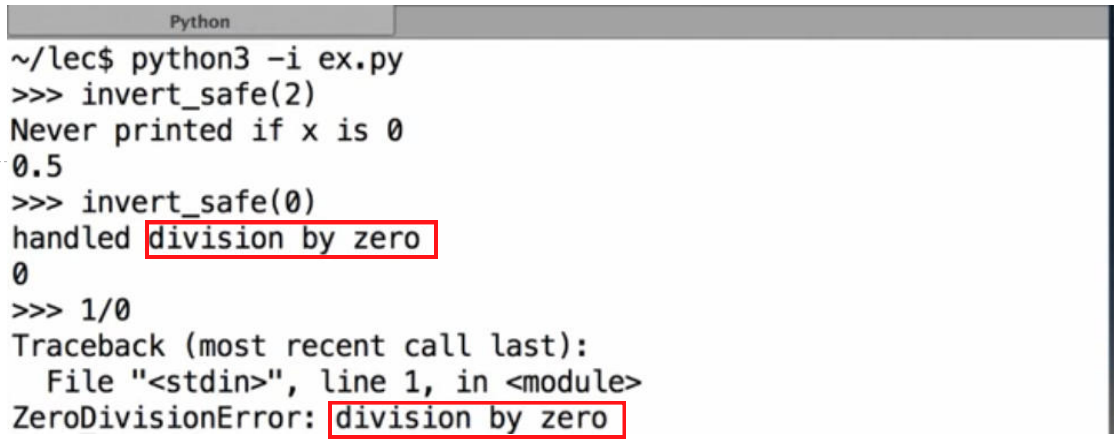
Exception handling can prevent a program from terminating

```
>>> try:
        x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0

handling a <class 'ZeroDivisionError'>
>>> x
0
```

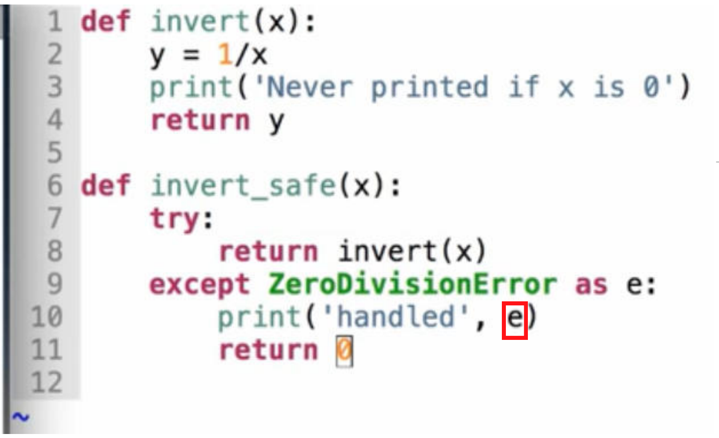
no matter what happens, x will be bound to something

Multiple try statements: Control jumps to the except suite of the most recent try statement that handles that type of exception

innermost



```
~/lec$ python3 -i ex.py
>>> invert_safe(2)
Never printed if x is 0
0.5
>>> invert_safe(0)
handled division by zero
0
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```



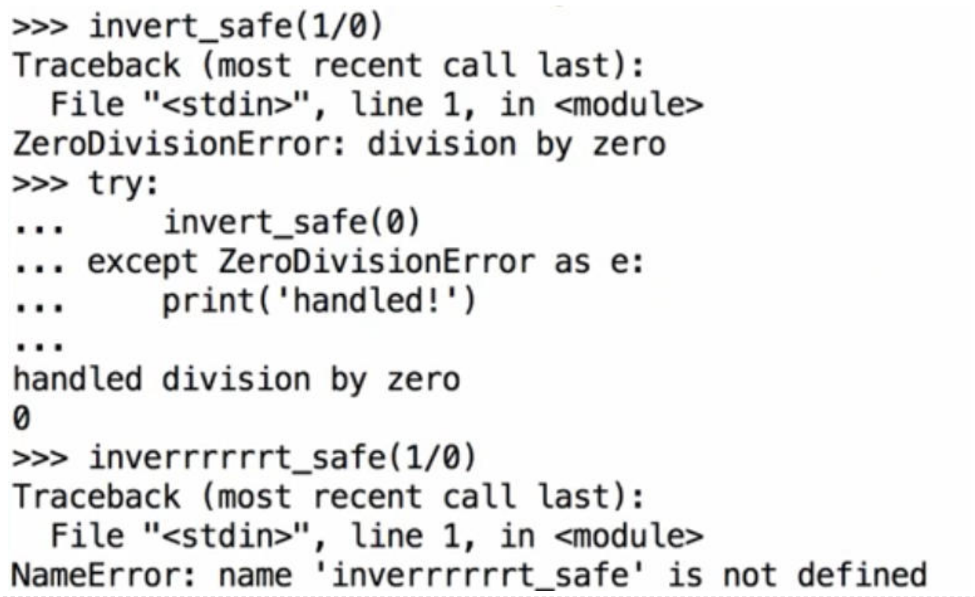
```
def invert(x):
    y = 1/x
    print('Never printed if x is 0')
    return y

def invert_safe(x):
    try:
        return invert(x)
    except ZeroDivisionError as e:
        print('handled', e)
        return 0

~
```

WWPD: What Would Python Display?

How will the Python interpreter respond?

```
def invert(x):
    inverse = 1/x  # Raises a ZeroDivisionError if x is 0
    print('Never printed if x is 0')
    return inverse

def invert_safe(x):
    try:
        return invert(x)
    except ZeroDivisionError as e:
        return str(e)
```

```
>>> invert_safe(1/0)
>>> try:
...     invert_safe(0)
... except ZeroDivisionError as e:
...     print('Hello!')
>>> inverrrrt_safe(1/0)
```



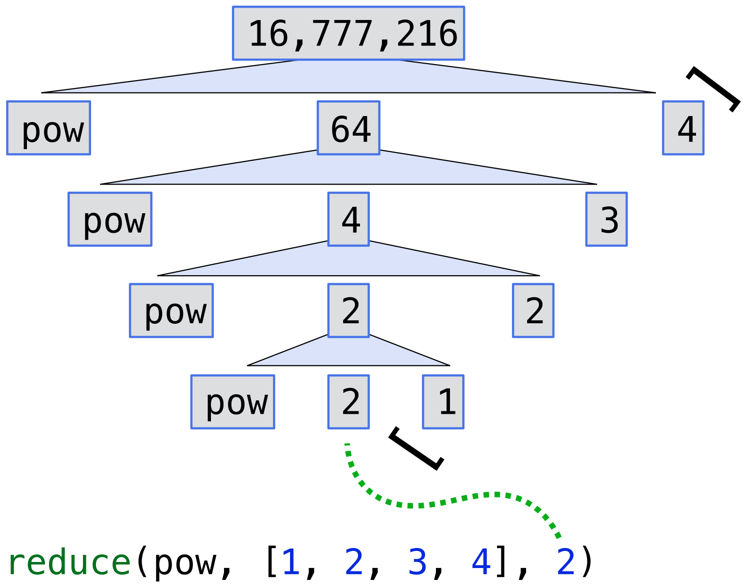
```
>>> invert_safe(1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> try:
...     invert_safe(0)
... except ZeroDivisionError as e:
...     print('handled!')
...
handled division by zero
0
>>> inverrrrrt_safe(1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'inverrrrrt_safe' is not defined
```

Example: Reduce

Reducing a Sequence to a Value

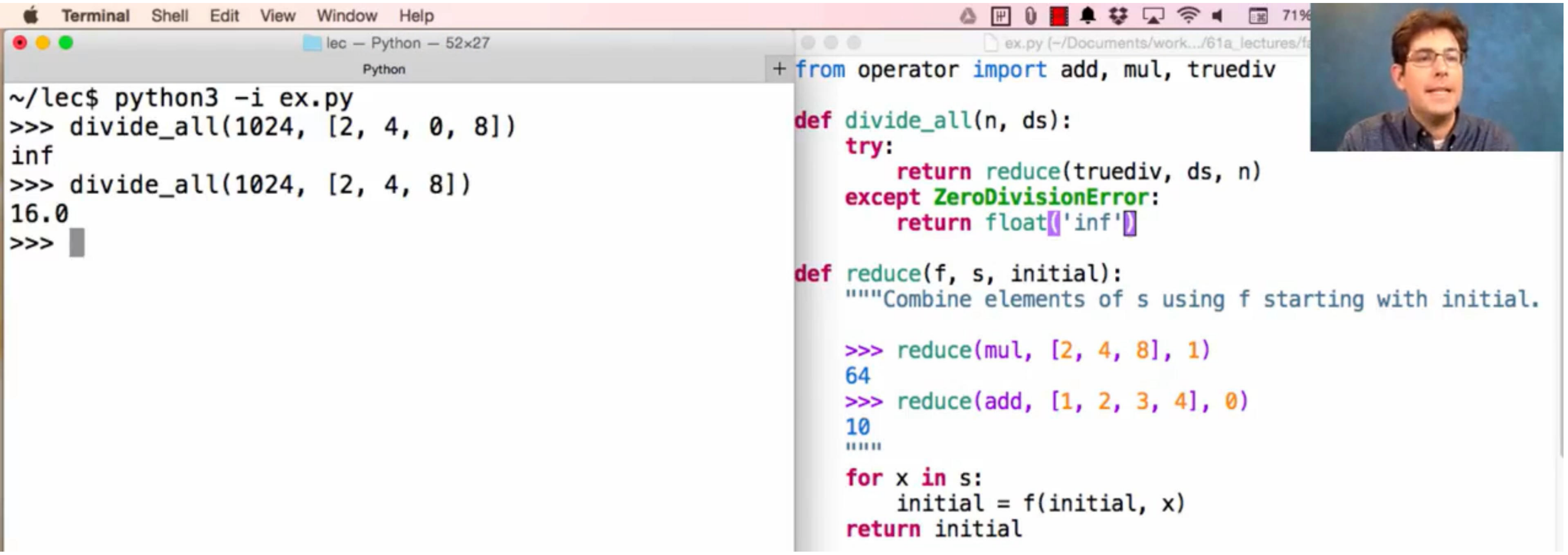
```
def reduce(f, s, initial):
    """Combine elements of s pairwise using f, starting with initial.

    E.g., reduce(mul, [2, 4, 8], 1) is equivalent to mul(mul(mul(1, 2), 4), 8).

    >>> reduce(mul, [2, 4, 8], 1)
    64
    """
```

```
f is ...
  a two-argument function
s is ...
  a sequence of values that can be the second argument
initial is ...
  a value that can be the first argument
```



```
reduce(pow, [1, 2, 3, 4], 2)
```

(Demo)


Terminal Shell Edit View Window Help
lec — Python — 52×27
Python
ex.py (~/Documents/work.../61a_lectures/f

```
~/lec$ python3 -i ex.py
>>> divide_all(1024, [2, 4, 0, 8])
inf
>>> divide_all(1024, [2, 4, 8])
16.0
>>> 
```

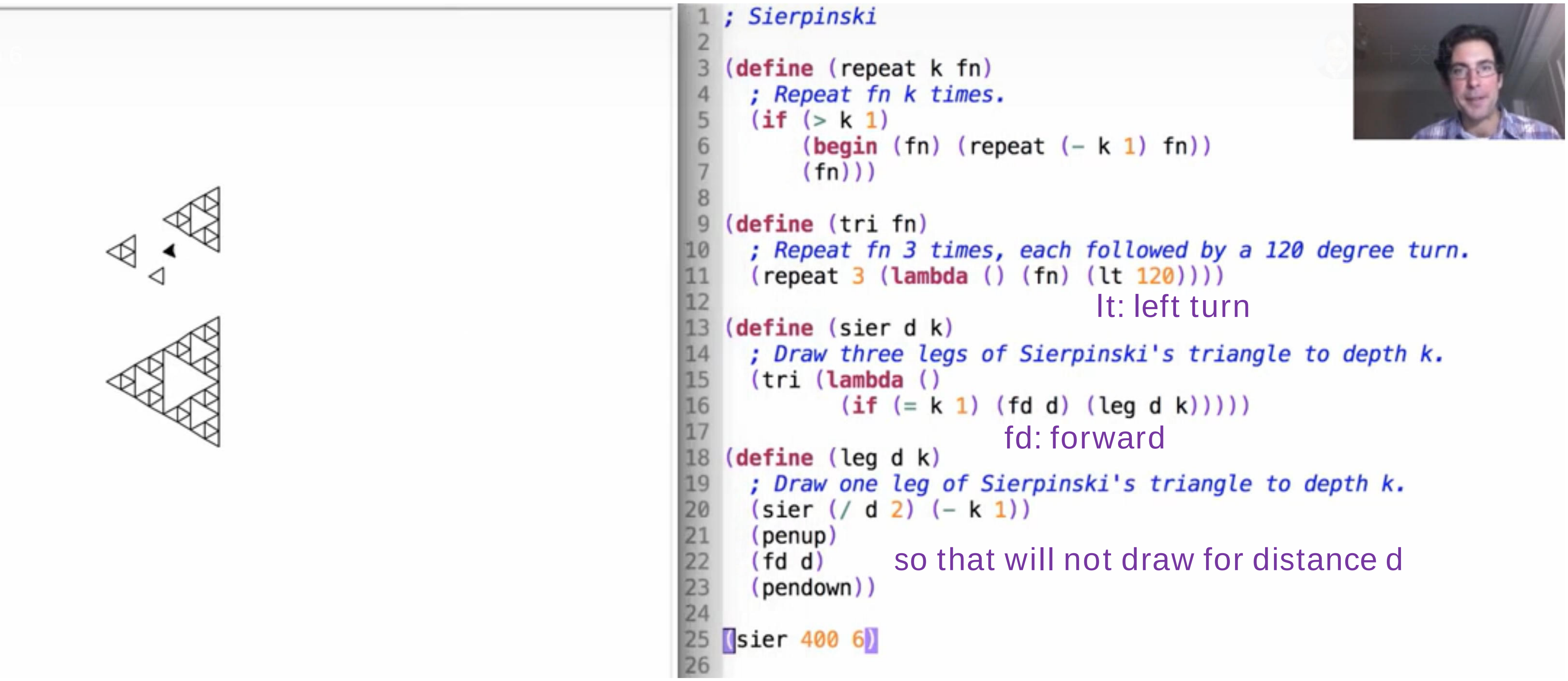
```python
from operator import add, mul, truediv

def divide_all(n, ds):
    try:
        return reduce(truediv, ds, n)
    except ZeroDivisionError:
        return float('inf')

def reduce(f, s, initial):
    """Combine elements of s using f starting with initial.

    >>> reduce(mul, [2, 4, 8], 1)
    64
    >>> reduce(add, [1, 2, 3, 4], 0)
    10
    """
    for x in s:
        initial = f(initial, x)
    return initial
```

Advantage of separating divide_all from reduce:

1. reduce doesn't have to know how to handle ZeroDivisionError, it only calculates
2. divide_all doesn't have to know how to calculate results, it only handles exception

Sierpinski's Triangle

(Demo)



```
; Sierpinski

(define (repeat k fn)
  ; Repeat fn k times.
  (if (> k 1)
      (begin (fn) (repeat (- k 1) fn))
      (fn)))

(define (tri fn)
  ; Repeat fn 3 times, each followed by a 120 degree turn.
  (repeat 3 (lambda () (fn) (lt 120))))

(define (sier d k)
  ; Draw three legs of Sierpinski's triangle to depth k.
  (tri (lambda ()
         (if (= k 1) (fd d) (leg d k)))))

(define (leg d k)
  ; Draw one leg of Sierpinski's triangle to depth k.
  (sier (/ d 2) (- k 1))
  (penup)
  (fd d)
  (pendown))

(sier 400 6)
```

lt: left turn

fd: forward

so that will not draw for distance d