Exceptions

Class outline:

- Scheme: Programs as data
- Python: Exceptions

Programs as data

A Scheme Expression is a Scheme List

Scheme programs consist of expressions, which can be:

- Primitive expressions: `2` `3.3` `#t` `+` `quotient`
- Combinations: `(quotient 10 2)` `(not #t)`

The built-in Scheme list data structure can represent combinations:

```
(list 'quotient 10 2)
```

```
(eval (list 'quotient 10 2))
```

In such a language, it is straightforward to write a program that writes a program.

A Scheme Expression is a Scheme List

Scheme programs consist of expressions, which can be:

- Primitive expressions: `2` `3.3` `#t` `+` `quotient`
- Combinations: `(quotient 10 2)` `(not #t)`

The built-in Scheme list data structure can represent combinations:

```
(list 'quotient 10 2)          ; (quotient 10 2)
```

```
(eval (list 'quotient 10 2))
```

In such a language, it is straightforward to write a program that writes a program.

A Scheme Expression is a Scheme List

Scheme programs consist of expressions, which can be:

- Primitive expressions: `2` `3.3` `#t` `+` `quotient`
- Combinations: `(quotient 10 2)` `(not #t)`

The built-in Scheme list data structure can represent combinations:

```
(list 'quotient 10 2)          ; (quotient 10 2)
```

```
(eval (list 'quotient 10 2))   ; 5
```

In such a language, it is straightforward to write a program that writes a program.

Quasiquotation

There are two ways to quote an expression:

- Quote: `'(a b)` => `(a b)`
- Quasiquote: `` `(a b) `` => `(a b)`

They are different because parts of a quasiquoted expression can be **unquoted** with `,`

```
(define b 4)
```

- Quote: `'(a ,(+ b 1))` => `(a (unquote (+ b 1))`
- Quasiquote: `` `(a ,(+ b 1)) `` => `(a 5)`

Generating code

Quasiquotation is particularly convenient for generating Scheme expressions:

```
(define (make-adder n) `(lambda (d) (+ d ,n)))
(make-adder 2)
```

Generating code

Quasiquotation is particularly convenient for generating Scheme expressions:

```
(define (make-adder n) `(lambda (d) (+ d ,n)))
(make-adder 2)         ; (lambda (d) (+ d 2))
```

Exceptions

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

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

Exceptions

An **exception** is a built-in mechanism in a programming language to declare and respond to "exceptional" conditions.

A program raises an exception when an error occurs.

If the exception is not handled, the program will stop running entirely.

But if a programmer can anticipate when exceptions might happen, they can include code for **handling the exception**, so that the program continues running.

Many languages include exception handling: C++, Java, Python, JavaScript, etc.

Exceptions in Python

Python raises an exception whenever a runtime error occurs.

How an unhandled exception is reported:

```
>>> 10/0
Traceback (most recent call last):
    File "<stdin>", line 1, in
ZeroDivisionError: division by zero
```

If an exception is not handled, the program stops executing immediately.

Types of exceptions

A few exception types and examples of buggy code:

Exception	Example
`OverflowError`	`pow(2.12, 1000)`
`TypeError`	`'hello'[1] = 'j'`
`IndexError`	`'hello'[7]`
`NameError`	`x += 5`
`FileNotFoundError`	`open('dsfdfd.txt')`

See full list in the exceptions docs.

The try statement

To handle an exception (keep the program running), use a `try` statement.

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

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.

Try statement example

```
try:
    quot = 10/0
except ZeroDivisionError as e:
    print('handling a', type(e))
    quot = 0
```

Try in PythonTutor

Try inside a function

```
def div_numbers(dividend, divisor):
    try:
        quotient = dividend/divisor
    except ZeroDivisionError:
        print("Function was called with 0 as divisor")
        quotient = 0
    return quotient

div_numbers(10, 2)
div_numbers(10, 0)
div_numbers(10, -1)
```

Try in PythonTutor

What would Python Do?

```
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:
        print('Handled', e)
        return 0
```

What would Python Do?

```
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:
        print('Handled', e)
        return 0
```

```
invert_safe(1/0)
```

What would Python Do?

```
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:
        print('Handled', e)
        return 0
```

```
invert_safe(1/0)
```

```
try:
    invert_safe(0)
except ZeroDivisionError as e:
    print('Handled!')
```

What would Python Do?

```
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:
        print('Handled', e)
        return 0
```

```
invert_safe(1/0)
```

```
try:
    invert_safe(0)
except ZeroDivisionError as e:
    print('Handled!')
```

```
inverrrrt_safe(1/0)
```

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
```

Raise statements

Any type of exception can be raised with a `raise` statement

```
raise <expression>
```

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

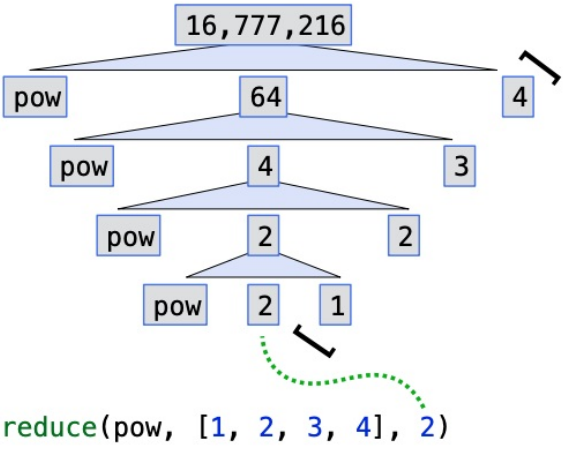
Exceptions are constructed like any other object. E.g., `TypeError('Bad argument!')`

Exercises

Exercise: Reduce

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

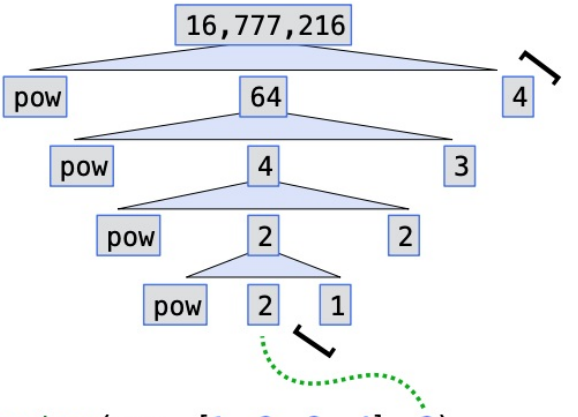
- `f`: a two-argument function
- `s`: a sequence of values that can be the second argument
- `initial`: a value that can be the first argument



Exercise: Reduce (Solution 1)

```
def reduce(f, s, initial):
    """Combine elements of s pairwise using f, starting with initial.
    >>> reduce(mul, [2, 4, 8], 1)
    64
    """
    if not s:
        return initial
    else:
        first, rest = s[0], s[1:]
        return reduce(f, rest, f(initial, first))
```

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

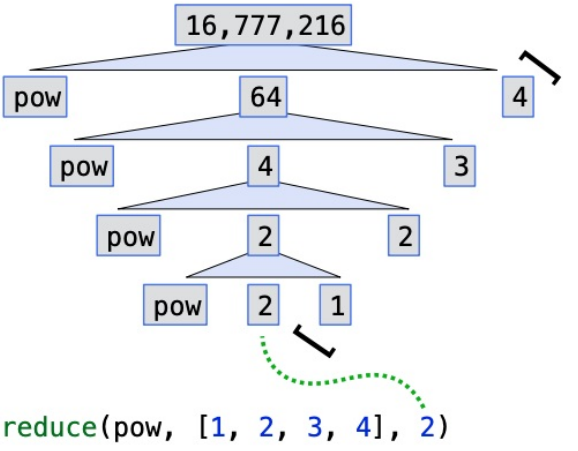



```
reduce(pow, [1, 2, 3, 4], 2)
```

Exercise: Reduce (Solution 2)

```
def reduce(f, s, initial):
    """Combine elements of s pairwise using f, starting with initial.
    >>> reduce(mul, [2, 4, 8], 1)
    64
    >>> reduce2(pow, [1, 2, 3, 4], 2)
    16777216
    """
    for x in s:
        initial = f(initial, x)
    return initial
```

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




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

Exercise: Divide all

```
def divide_all(n, ds):
    """Divide n by every d in ds.

    >>> divide_all(1024, [2, 4, 8])
    16.0
    >>> divide_all(1024, [2, 4, 0, 8])
    inf
    """
```

Use the `reduce()` function we just defined...

Exercise: Divide all (Solution)

```
def divide_all(n, ds):
    """Divide n by every d in ds.

    >>> divide_all(1024, [2, 4, 8])
    16.0
    >>> divide_all(1024, [2, 4, 0, 8])
    inf
    """
    try:
        return reduce(truediv, ds, n)
    except ZeroDivisionError:
        return float('inf')
```

Using the `reduce()` function we just defined.