Today's Topic: Handling Errors

Sometimes, computers don't do exactly what we expect

- A function receives unexpected argument types
- Some resource (such as a file) is not available
- A network connection is lost

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 a crash

Unhandled exceptions will cause Python to halt execution

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.

However, exception handling tends to be slow.
Assert Statements

Assert statements raise an exception of type AssertionError

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

Assertions are designed to be used liberally and then disabled in "production" systems. "0" stands for optimized.

```bash
python3 -O
```

Whether assertions are enabled is governed by a bool `__debug__`

Demo
Raise Statements

Exceptions are raised with a raise statement.

```
raise <expression>
```

<expression> must evaluate to an exception instance or class.

Exceptions are constructed like any other object; they are just instances of classes that inherit from BaseException.

**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
Try Statements

Try statements handle exceptions

```python
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

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

Demo
How will the Python interpreter respond?

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

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('Handled!')

>>> inverrrrrt_safe(1/0)
```
Reading Scheme Lists

A Scheme list is written as elements in parentheses:

((element_0> element_1> ... element_n>)

Each <element> can be a combination or primitive.

(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))

The task of parsing a language involves coercing a string representation of an expression to the expression itself.

Parsers must validate that expressions are well-formed.

Demo (http://inst.eecs.berkeley.edu/~cs61a/fa12/projects/scalc/scheme_reader.py.html)
Parsing

A Parser takes a sequence of lines and returns an expression.

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time

- Tree-recursive process
- Balances parentheses
- Returns tree structure
- Processes multiple lines
Recursive Syntactic Analysis

A predictive recursive descent parser inspects only $k$ tokens to decide how to proceed, for some fixed $k$.

Can English be parsed via predictive recursive descent?

*The horse raced past the barn fell.*

(sentence subject)

(ridden (that was)
Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression.

'(', '+', 1, '(', '-', 23, ')', '*', 4, 5.6, ')', '

Recursive call: scheme_read sub-expressions and combine them

Base case: symbols and numbers

Demo (http://inst.eecs.berkeley.edu/~cs61a/fa12/projects/scalc/scheme_reader.py.html)