Announcements
Pairs Review
Pairs and Lists
Pairs and Lists

In the late 1950s, computer scientists used confusing names
Pairs and Lists

In the late 1950s, computer scientists used confusing names
• **cons**: Two-argument procedure that creates a pair
Pairs and Lists

In the late 1950s, computer scientists used confusing names

- **cons**: Two-argument procedure that creates a pair

(cons 1 2)
Pairs and Lists

In the late 1950s, computer scientists used confusing names

- **cons**: Two-argument procedure that creates a pair

\[
\text{cons}(1, 2) = (1, 2)
\]
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

\[(\text{cons } 1 \ 2) \quad \begin{array}{c}1 \ 2 \end{array} \quad (1 \ . \ 2)\]
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

(cons 1 2)  
(1 . 2)
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

(cons 1 2)    1 2    (1 . 2)
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
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

\[
\begin{align*}
\text{(cons 1 2)} & \quad (1 \ . \ 2) \\
\text{(cons 2 nil)} & \quad (1 \ . \ 2)
\end{align*}
\]
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

```scheme
(cons 1 2)  (cons 2 nil)
```

```
1 2
```

```
2 nil
```

```
(1 . 2)
```

```
2
```

```
nil
```
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

\[
(\text{cons} \ 1 \ 2) \quad (1 \ . \ 2) \\
(\text{cons} \ 2 \ \text{nil}) \\
(\text{cons} \ 2 \ \text{nil})
\]
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
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

```
(cons 1 2)  \[\text{1 2}\]  \(1 . 2\)
(cons 2 nil)  \[\text{2 nil}\]  \(2 \rightarrow \text{nil}\)
(cons 1 (cons 2 nil))  \[\text{1 \(\rightarrow\) \[2 \rightarrow \text{nil}\]}\]
Pairs and Lists

In the late 1950s, computer scientists used confusing names
• **cons**: Two-argument procedure that creates a pair
  (cons 1 2)  
  \[ \text{1} \rightarrow \text{2} \]

• **car**: Procedure that returns the first element of a pair
  (cons 2 nil)  
  \[ \begin{array}{c} \text{2} \\ \hline \end{array} \rightarrow \text{nil} \]

• **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

\[(\text{cons 1 (cons 2 nil)})\]

\[\begin{array}{c} \text{1} \\ \hline \text{2} \\ \hline \text{nil} \end{array}\]
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
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 (cons 2 nil))
Pairs and Lists

In the late 1950s, computer scientists used confusing names:

• **cons**: Two-argument procedure that creates a pair
  
  ![Cons Diagram]

• **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.

```scheme
> (define x (cons 1 2))
```

![List Diagram]
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

```scheme
> (define x (cons 1 2))
> x
```

```
(cons 1 2) 1 2
(cons 2 nil) 2 nil
(cons 2 nil) 2
(1 . 2)
```
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

```scheme
> (define x (cons 1 2))
> x
(1 . 2)
```

(cons 1 2)  1 2  (1 . 2)
(cons 2 nil)  2  nil
(cons 2 nil)  2  nil
(cons 1 (cons 2 nil))  1  2  nil
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

```scheme
> (define x (cons 1 2))
> x
(1 . 2)
> (car x)
```

```scheme
(cons 1 (cons 2 nil))
```
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.

```scheme
> (define x (cons 1 2))
> x
(1 . 2)
> (car x)
1

(cons 1 2)  (cons 2 nil)

(1 . 2)  2

(cons 1 (cons 2 nil))

1 2  nil
```
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.

```scheme
> (define x (cons 1 2))
> x
(1 . 2)
> (car x)
1
> (cdr x)
```
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

> (define x (cons 1 2))
> x
(1 . 2)
> (car x)
1
> (cdr x)
2
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

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

```
> (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)
```
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

```
> (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)
```
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

```scheme
> (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)
```

(Demo)
Scheme Lists and Quotation
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair.
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair.

```scheme
> (cdr (cdr '(1 2 . 3)))
```
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair.

```scheme
> (cdr (cdr '(1 2 . 3))
3
```
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair.

```scheme
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists.
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair.

> (cdr (cdr '(1 2 . 3)))
3

However, dots appear in the output only of ill-formed lists.

> '(1 2 . 3)
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair.

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists.

```
> '(1 2 . 3)
```

1 ← 2 3
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair.

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists.

```
> '(1 2 . 3)  
(1 2 . 3)
```

```
  1  2  3
```
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair.

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists.

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
```

![Diagram of list structure]
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair.

> (cdr (cdr '(1 2 . 3)))
3

However, dots appear in the output only of ill-formed lists.

> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair.

> (cdr (cdr '(1 2 . 3)))
3

However, dots appear in the output only of ill-formed lists.

> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))

![Diagram of list structures]
Dots can be used in a quoted list to specify the second element of the final pair.

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists.

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
```

![Diagram of list operations](image-url)
Dots can be used in a quoted list to specify the second element of the final pair.

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists.

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
```
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair.

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists.

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
1 2 3 4
```

```
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
1 2 3 4 5
```
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair.

```scheme
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists.

```scheme
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
(1 2 3)
```
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair.

> (cdr (cdr '(1 2 . 3)))
3

However, dots appear in the output only of ill-formed lists.

> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
(1 2 3)

What is the printed result of evaluating this expression?
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair.

```scheme
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists.

```scheme
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
(1 2 3)
```

What is the printed result of evaluating this expression?

```scheme
> (cdr '((1 2) . (3 4 . (5))))
```
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair.

> (cdr (cdr '(1 2 . 3)))
3

However, dots appear in the output only of ill-formed lists.

> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
(1 2 3)

What is the printed result of evaluating this expression?

> (cdr '((1 2) . (3 4 . (5))))
(3 4 5)
Exceptions
Today's Topic: Handling Errors
Today's Topic: Handling Errors

Sometimes, computer programs behave in non-standard ways
Today's Topic: Handling Errors

Sometimes, computer programs behave in non-standard ways

- A function receives an argument value of an improper type
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
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
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

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

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

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.
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.
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

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.
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
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.
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 Statements

Assert statements raise an exception of type AssertionError

```
assert <expression>, <string>
```
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.
Assert Statements

Assert statements raise an exception of type AssertionError

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

Assertions are designed to be used liberally. They can be ignored to increase efficiency by running Python with the "-O" flag; "0" stands for optimized

```bash
python3 -O
```
**Assert Statements**

Assert statements raise an exception of type AssertionError

```python
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

```bash
python3 -O
```

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

Exceptions are raised with a raise statement
Raise Statements

Exceptions are raised with a raise statement

raise <expression>
Raise Statements

Exceptions are raised with a raise statement

```
raise <expression>
```

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

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!')`
Raise Statements

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
Raise Statements

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
Raise Statements

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
Raise Statements

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
Raise Statements

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

Try statements handle exceptions
Try Statements

Try statements handle exceptions

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

Try statements handle exceptions

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

Execution rule:
Try Statements

Try statements handle exceptions

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

Execution rule:

The <try suite> is executed first
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
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
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
Handling Exceptions

Exception handling can prevent a program from terminating
Handling Exceptions

Exception handling can prevent a program from terminating

>>> try:
Handling Exceptions

Exception handling can prevent a program from terminating

```python
>>> try:
    x = 1/0
```
Handling Exceptions

Exception handling can prevent a program from terminating

```python
>>> try:
    x = 1/0
    except ZeroDivisionError as e:
```
Handling Exceptions

Exception handling can prevent a program from terminating

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

Exception handling can prevent a program from terminating

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

Exception handling can prevent a program from terminating

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

handling a <class 'ZeroDivisionError'>
```
Handling Exceptions

Exception handling can prevent a program from terminating

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

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

14
Handling Exceptions

Exception handling can prevent a program from terminating

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

handling a <class 'ZeroDivisionError'>

```
Handling Exceptions

Exception handling can prevent a program from terminating

```python
>>> 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
Handling Exceptions

Exception handling can prevent a program from terminating

```python
>>> 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)
WWPD: What Would Python Do?

How will the Python interpreter respond?
WWPD: What Would Python Do?

How will the Python interpreter respond?
WWPD: What Would Python Do?

How will the Python interpreter respond?

```python
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)
```
**WWPD: What Would Python Do?**

How will the Python interpreter respond?

```python
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)
```

```
WWPD?
```
**WWPD: What Would Python Do?**

How will the Python interpreter respond?

```python
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:
```
**WWPD: What Would Python Do?**

How will the Python interpreter respond?

```python
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)
```

---

15
**WWPD: What Would Python Do?**

How will the Python interpreter respond?

```python
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)
```

```python
>>> invert_safe(1/0)
>>> try:
...    invert_safe(0)
... except ZeroDivisionError as e:
```
**WWPD: What Would Python Do?**

How will the Python interpreter respond?

```python
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('Handled!')
```
**WWPD: What Would Python Do?**

How will the Python interpreter respond?

```python
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('Handled!')
>>> inverrrrrt_safe(1/0)
```
Example: Reduce
Reducing a Sequence to a Value
Reducing a Sequence to a Value

```python
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
    """
```
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
Reducing a Sequence to a Value

```python
def reduce(f, s, initial):
    """Combine elements of s pairwise using f, starting with initial."
    E.g., \(\text{reduce}(\text{mul}, [2, 4, 8], 1)\) is equivalent to \(\text{mul}(\text{mul}(1, 2), 4), 8)\).

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

\(f\) is ...

\(a\ two\text{-argument\ function}\)

\(s\) is ...

\(a\ sequence\ of\ values\ that\ can\ be\ the\ second\ argument\)
Reducing a Sequence to a Value

```python
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
Reducing a Sequence to a Value

```python
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)
Reducing a Sequence to a Value

```python
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*

```python
reduce(pow, [1, 2, 3, 4], 2)
```
Reducing a Sequence to a Value

```python
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
```

```python
reduce(pow, [1, 2, 3, 4], 2)
```
Reducing a Sequence to a Value

```python
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*

```python
reduce(pow, [1, 2, 3, 4], 2)
```
Reducing a Sequence to a Value

```python
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
```

```python
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
```

```python
reduce(pow, [1, 2, 3, 4], 2)
```
Reducing a Sequence to a Value

```python
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
    ""
```

\[ f \text{ is ...} \]
\[ \text{a two-argument function} \]
\[ s \text{ is ...} \]
\[ \text{a sequence of values that can be the second argument} \]
\[ \text{initial is ...} \]
\[ \text{a value that can be the first argument} \]

reduce(pow, [1, 2, 3, 4], 2)
Reducing a Sequence to a Value

```python
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)
Reducing a Sequence to a Value

```python
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)
```
Reducing a Sequence to a Value

```python
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)
```
Reducing a Sequence to a Value

```python
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*

```python
reduce(pow, [1, 2, 3, 4], 2)
```
Reducing a Sequence to a Value

```python
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)
```
Reducing a Sequence to a Value

```python
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)
```
Reducing a Sequence to a Value

```python
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

\( \text{initial} \) is ...
\( a \) value that can be the first argument

\[
\text{reduce}(\text{pow}, [1, 2, 3, 4], 2)
\]
Reducing a Sequence to a Value

```python
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*
Reducing a Sequence to a Value

```python
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*

```python
reduce(pow, [1, 2, 3, 4], 2)
```
Reducing a Sequence to a Value

```python
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

(Demo)
Sierpinski's Triangle

(Demo)