Functional Abstraction
Announcements
Lambda Function Environments
A lambda function's parent is the current frame in which the lambda expression is evaluated

```python
a = 1
def f(g):
    a = 2
    return lambda y: a * g(y)
f(lambda y: a + y)(a)
```
Return Statements

A return statement completes the evaluation of a call expression and provides its value:

- f(x) for user-defined function f: switch to a new environment; execute f's body
- return statement within f: switch back to the previous environment; f(x) now has a value

Only one return statement is ever executed while executing the body of a function

```python
def end(n, d):
    """Print the final digits of N in reverse order until D is found."

>>> end(34567, 5)
7
6
5

    while n > 0:
        last, n = n % 10, n // 10
        print(last)
    if d == last:
        return None
```

(Demo)
Control
If Statements and Call Expressions

Let's try to write a function that does the same thing as an if statement.

Execution Rule for Conditional Statements:
Each clause is considered in order.
1. Evaluate the header's expression (if present).
2. If it is a true value (or an else header), execute the suite & skip the remaining clauses.

Evaluation Rule for Call Expressions:
1. Evaluate the operator and then the operand subexpressions
2. Apply the function that is the value of the operator to the arguments that are the values of the operands

```python
def if_(c, t, f):
    if c:
        return t
    else:
        return f
```
Control Expressions
Logical Operators

To evaluate the expression `<left> and <right>`:

1. Evaluate the subexpression `<left>`. 
2. If the result is a false value v, then the expression evaluates to v. 
3. Otherwise, the expression evaluates to the value of the subexpression `<right>`. 

To evaluate the expression `<left> or <right>`:

1. Evaluate the subexpression `<left>`. 
2. If the result is a true value v, then the expression evaluates to v. 
3. Otherwise, the expression evaluates to the value of the subexpression `<right>`. 

(Demo)
Abstraction
Functional Abstractions

```python
def square(x):
    return mul(x, x)

def sum_squares(x, y):
    return square(x) + square(y)
```

What does `sum_squares` need to know about `square`?

- Square takes one argument.  
  Yes
- Square has the intrinsic name `square`.  
  No
- Square computes the square of a number.  
  Yes
- Square computes the square by calling `mul`.  
  No

```python
def square(x):
    return pow(x, 2)

def square(x):
    return mul(x, x-1) + x
```

If the name “square” were bound to a built-in function, `sum_squares` would still work identically.
Choosing Names

Names typically don’t matter for correctness

**but**

they matter a lot for composition

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>true_false</td>
<td>rolled_a_one</td>
</tr>
<tr>
<td>d</td>
<td>dice</td>
</tr>
<tr>
<td>helper</td>
<td>take_turn</td>
</tr>
<tr>
<td>my_int</td>
<td>num_rolls</td>
</tr>
<tr>
<td>l, I, 0</td>
<td>k, i, m</td>
</tr>
</tbody>
</table>

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (**print**), their behavior (**triple**), or the value returned (**abs**).
Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

```python
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```

```python
hypotenuse = sqrt(square(a) + square(b))
if hypotenuse > 1:
    x = x + hypotenuse
```

Meaningful parts of complex expressions:

```python
x1 = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```python
discriminant = square(b) - 4 * a * c
x1 = (-b + sqrt(discriminant)) / (2 * a)
```

More Naming Tips

- Names can be long if they help document your code:
  ```python```
  ```
  average_age = average(age, students)
  ```
  ```
  is preferable to
  ```
  ```
  # Compute average age of students
  aa = avg(a, st)
  ```

- Names can be short if they represent generic quantities: counts, arbitrary functions, arguments to mathematical operations, etc.

  ```
  n, k, i - Usually integers
  x, y, z - Usually real numbers
  f, g, h - Usually functions
  ```
Errors & Tracebacks
## Taxonomy of Errors

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax Errors</td>
<td>Detected by the Python interpreter (or editor) before the program executes</td>
</tr>
<tr>
<td>Runtime Errors</td>
<td>Detected by the Python interpreter while the program executes</td>
</tr>
<tr>
<td>Logic &amp; Behavior Errors</td>
<td>Not detected by the Python interpreter; what tests are for</td>
</tr>
</tbody>
</table>

(Demo)