Abstraction

Functional Abstractions

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

```python
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

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>true_false</td>
<td>rolled_a_dice</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>1, 2, 0</td>
<td>k, l, m</td>
</tr>
</tbody>
</table>

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

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))
  ```

- Meaningful parts of complex expressions:
  ```python
  x1 = (-b + sqrt(discriminant)) / (2 * a)
  ```

More Naming Tips

- Names should be long if they help you document your code:
  ```python
  average_age = average(age, students)
  ```

- Names should be short if they represent generic quantities: counts, arbitrary functions, arguments to mathematical operations, etc.
  ```python
  n, k, i - Usually integers
  x, y, z - Usually real numbers
  f, g, h - Usually functions
  ```

Test-Driven Development

Write the test of a function before you write the function.

A test will clarify the domain, range, & behavior of a function.

Tests can help identify tricky edge cases.

Develop incrementally and test each piece before moving on.

You can’t depend upon code that hasn’t been tested.

Run your old tests again after you make new changes.

Bonus idea: Run your code interactively.

Don’t be afraid to experiment with a function after you write it.

Interactive sessions can become doctests. Just copy and paste.
Currying

Function Currying

```python
def make_adder(n):
    return lambda k: n + k
```

```python
>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

Curry: Transform a multi-argument function into a single-argument, higher-order function

Decorators

Function Decorators

```python
@trace1
def triple(x):
    return 3 * x
```

is identical to

```python
def triple(x):
    return 3 * x
triple = trace1(triple)
```

Why not just use this?

Review

```python
def delay(arg):
    print('delayed')
def g():
    return arg
return g
```

What Would Python Display?

The `print` function returns `None`. It also displays its arguments (separated by spaces) when it is called.

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>print(5)</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>print(print(5))</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>delay(delay)</td>
<td>delayed</td>
<td>delayed 6</td>
</tr>
<tr>
<td>delay(print)</td>
<td>delayed</td>
<td>delayed 4</td>
</tr>
</tbody>
</table>

Interactive Output

```python
print(5)
n=delay(print)()
n()
```

None

None

None

5

5

None

This expression

Return Value

Return Value

Global frame

horse

mask

func horse(mask)

[ parent = Global ]

func

λ

[ parent = Global ]

def horse(mask):
    return horse(mask)

mask = lambda horse: horse(2)
horse(mask)