Function Examples

Hog Contest Rules
• Up to two people submit one entry;
  Max of one entry per person
• Slight rule changes
• Your score is the number of entries
  against which you win more than
  50.00001% of the time
• Strategies are time-limited
• All strategies must be deterministic,
  pure functions of the players’ scores
• All winning entries will receive
  extra credit
• The real prize: honor and glory
• See website for detailed rules

Hog Contest Winners

Abstraction

Choosing Names

Functional Abstractions

Which Values Deserve a Name

From: true_false d helper my_int 1, 1, 0
To: rolled_a_one dice take_turn num_rolls a, i, m

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).

Reasons to add a new name
• Repeated compound expressions:
  if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
  x = x + hypotenuse
  if hypotenuse > 1:
    x = x + hypotenuse

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

  x = x + sqrt(square(a) + square(b))

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

Meaningful parts of complex expressions:
• a = [-b + sqrt(square(b) - 4 * a * c)] / (2 * a)
• discriminant = square(b) - 4 * a * c
  a = (-b - sqrt(discriminant)) / (2 * a)

Types of names:
• Names can be long if they help document your code:
  average_age = average(age, students)
• 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
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.

Test-Driven Development

Function Currying

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

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

There's a general relationship between these functions

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

Function Decorators

```
@trace
def triple(x):
    return 3 * x
```

is identical to

```
def triple(x):
    return 3 * x
triple = trace(triple)
```

Why not just use this?

Function decorator

What Would Python Display?

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

```
from operator import add, mul

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

```
Names in nested def statements can refer to their enclosing scope
```

This expression | Evaluates to | Interactive Output
-----------------|-------------|------------------
5                | 5           | 5
print(5)         | None        | 5
print(print(5))  | None        | 5
None             | None        | None
None             | None        | None
delay()          | | delayed 6
print(delay())    | 6           | delayed 6
print(print())    | None        | delayed 4
print(print())    | None        | None
``
def horse(mask):
    horse = mask
    return horse

return horse(mask)

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