Abstracting with Functions

Review: Abstracting with Layers

We’ve seen that layers are a means of abstraction. We can populate a fishtank by cloning and transforming a single prototype fish pattern expressed as a layer:

```python
# Add pink hat *before* any clones are made
hat = Polygon(Point(-23,-37),Point(9,-31),
               Point(37,-50),Point(25,-20),
               Point(-10,-13))
hat.setFillColor('pink')
fish.add(hat)
```

Review: Drawbacks of Layers

Although Layers are powerful, they do not let us abstract over all the properties of our fish that we might want to change.

- What if we want different fish to have different body or tail colors?
- What if we want different fish to have larger or smaller eyes?

We cannot express these differences with Layers. Why not?

But we can express them with user-defined functions, a more powerful abstraction mechanism that we will study in this lecture.

Functions

Functions take inputs and return outputs based on those inputs.

Here are examples of built-in functions you have seen:

| In [...]                | Out [...]
|------------------------|-----------
| max(7, 3)              | 7         |
| min(7, 3, 2, 9)        | 2         |
| type(123)              | int       |
| len('CS111')           | 5         |
| str(4.0)               | '4.0'     |
| int(-2.978)            | -2        |
| float(42)              | 42.0      |
| round(2.718, 1)        | 2.7       |
Some functions perform actions instead of returning outputs

These actions are called **side effects**.

For example, displaying text in the interactive console (Canopy’s Python pane) is a side effect of the `print` and `help` functions:

In [1]: print("The max value is: "+str(max(23,78)))
The max value is: 78

In [2]: help(max)
Help on built-in function max in module __builtin__:

max(...)  
max(iterable[, key=func]) -> value  
max(a, b, c, ...[, key=func]) -> value

Anatomy of a User-defined Function

Functions are a way of abstracting over computational processes by capturing common patterns.

**Function definitions**

```
def square(x):
    return x * x
```

- **Definition**
  - Parameter
  - Header
  - Body
  - Body is indented!
  - Keyword indicating return value. Always ends execution of function!

**Function calls/invocations**

```
square(5)  
square(10)  
square(-3)  
```

- **Calls**
  - Arguments
  - Results

A function can be called many times.

Function diagrams summarize what functions do

Parameters

A parameter names “holes” in the body that will be filled in with the *argument value* for each invocation.

The particular name we use for a parameter is irrelevant, as long as we use the name consistently in the body.
**Python Function Call Model**

We need a model to understand how function calls work.

```python
def square(x):
    return x * x
```

**Step 1:**
- **Square Function:**
  - `square(2 + 3)`
  - `square(5)`

**Step 2:**
- **Function Call Frame:**
  1. Create a function call frame with
     - (1) a variable box named by each parameter and filled with the corresponding argument value; and
     - (2) the body expression(s) from the function definition.

```
x = 5
square frame
```

**Step 3:**
- Evaluate the body expression(s), using the values in the parameter variable boxes any time a parameter is referenced.
  - Do you see why parameter names don’t matter as long as they’re consistent?

```
x = 5
return 25
```

**Step 4:**
- The frame is discarded after the value returned by the frame “replaces” the call.

```
return 5 * 5
```

```
x = 5
return 25
```

**Multiple parameters**

A function can take as many parameters as needed. They are separated via comma.

```python
def energy(m, v):
    """Calculate kinetic energy"""
    return 0.5 * m * v**2
```

```python
def pyramidVolume(len, wid, hgh):
    """Calculate volume rectangular pyramid"""
    return (len * wid * hgh) / 3.0
```

```python
import math
def distanceBetweenPoints(x1, y1, x2, y2):
    """Calculate the distance between points"""
    return math.sqrt((x2-x1)**2 + (y2-y1)**2)
```

**Output of a function:**

- **Return** specifies the result of the function invocation
- **Print** causes characters to be displayed in the shell.

```python
def square(x):
    return x*x

def squarePrintArg(x):
    print('The argument of square is ' + str(x))
    return x*x
```

```
In [2]: square(3) + square(4)
Out[2]: 25
```

```
In [3]: squarePrintArg(3) + squarePrintArg(4)
The argument of square is 3
The argument of square is 4
Out[3]: 25
```
Don’t confuse \texttt{return} with \texttt{print}!

```python
def printSquare(a):
    print('square of \texttt{'} + str(a) + \texttt{'} is \texttt{'} + str(square(a))
```

In [4]: printSquare(5)
     square of 5 is 25
In [5]: printSquare(3) + printSquare(4)
     square of 3 is 9
     square of 4 is 16

---

```
TypeError Traceback (most recent call last)
<ipython-input-10-ff81dee8cf8f> in <module>()
      1 printSquare(3) + printSquare(4)
```

\texttt{printSquare} does not return a number, so it doesn’t make sense to add the two invocations!

---

**Examples: Function with side-effect and no return value**

```python
def printBanner(s):
    # 5 stars, 3 spaces, input string, 3 spaces, 5 stars
    banner_length = 5 + 3 + len(s) + 3 + 5
    print('*' * banner_length)
    print('*****' + ' ' + s + ' ' + '*****')
    print('*' * banner_length)
```

printBanner('CS111')

```
********************
***** CS111 *****
********************
```

printBanner('Pied Piper')

```
********************
***** Pied Piper *****
********************
```

---

**Local variables**

An assignment to a variable within a function definition creates/changes a local variable.

Local variables exist only within a function’s body. They cannot be referred outside of it.

Parameters are also local variables that are assigned a value when the function is invoked. They also cannot be referred outside the function.

---

```python
def rightTrianglePerim(a, b):
    c = hypotenuse(a, b)
    return a + b + c
```

In [1]: rightTrianglePerim(3, 4)
     Out [1]: 12.0

In [2]: c
     NameError: name 'c' is not defined

In [3]: a
     NameError: name 'a' is not defined

In [4]: b
     NameError: name 'b' is not defined

---

**Example: Seconds to Days**

```python
def printTimeFromSeconds(s):
    # Total seconds
    seconds = s % 60 # Remaining seconds
    m = s / 60 # Total minutes
    minutes = m % 60 # Remaining minutes
    h = m / 60 # Total hours
    hours = h % 24 # Remaining hours
    days = h / 24 # Total days
    print(str(s) + ' seconds is equivalent to:')
    print(str(days) + ' days')
    print(str(hours) + ' hours')
    print(str(minutes) + ' minutes')
    print(str(seconds) + ' seconds')
```

In [1]: printTimeFromSeconds(1000000)
     1000000 seconds is equivalent to:
     11 days
     13 hours
     46 minutes
     40 seconds

---

Functions 4-13

Functions 4-15

Functions 4-14

Functions 4-16
Calling other functions

Functions can call other functions:

```python
import math

def hypotenuse(a, b):
    return math.sqrt(square(a) + square(b))

hypotenuse(3, 4) ➞ 5.0
hypotenuse(1, 1) ➞ 1.4142135623730951

def distanceBetweenPoints(x1, y1, x2, y2):
    # "Calculate the distance between points"
    return hypotenuse(x2-x1, y2-y1)
```

Function call model for `hypotenuse(3,4)` [1]

```
hypotenuse frame

a 3 b 4

return math.sqrt(square(a) + square(b))

square frame

x 4

return x*x

return 16

return 5.0
```

Function call model for `hypotenuse(3,4)` [2]

```
hypotenuse frame

a 3 b 4

return math.sqrt(square(a) + square(b))

square frame

x 3

return x*x

return 3*3

return 9
```

Function call model for `hypotenuse(3,4)` [3]

```
hypotenuse frame

a 3 b 4

return math.sqrt(square(a) + 16)
```

```
hypotenuse frame

a 3 b 4

return math.sqrt(25)
```

```
hypotenuse frame

a 3 b 4

return 5.0
```
Function Abstraction: Fishtank Revisited

We cannot make these fish by cloning a fish layer. Why?

```python
# Create a fishtank, and add three fish
tank = Canvas(600,400, 'skyBlue', 'FishFunctionWorld')
fish1 = makeFish()
tank.add(fish1)
fish1.moveTo(150, 100)
fish2 = makeFish()
tank.add(fish2)
fish2.moveTo(450, 150)
fish3 = makeFish()
tank.add(fish3)
fish3.moveTo(200, 300)
```

This makes a new fish Layer via a function call rather than a clone. With parameters (see next few slides), functions are more powerful than clones.

```python
def makeFish():
    fish = Layer()  # fish layer
    # body of the fish
    body = Ellipse(100,50,Point(0,0))
    body.setFillColor('yellow')
    fish.add(body)
    # green tail of the fish
    tail = Polygon()
    tail.addPoint(Point(-50,0))
    tail.addPoint(Point(-75,25))
    tail.addPoint(Point(-75,-25))
    tail.setFillColor('green')
    fish.add(tail)
    # black eye of the fish
    eye = Circle(5,Point(25,-5))
    eye.setFillColor('black')
    fish.add(eye)
return fish
```

```
# Zero-Parameter Functions

Sometimes it's helpful to define/use functions that have zero parameters. Note: you still need parentheses after the function name when defining and invoking the function.

```python
def rocks():
    print('CS111 rocks!')
def rocks3():
    rocks()
    rocks()
    rocks()
```

Python libraries have useful built-in functions with zero parameters and a return value:

```python
import random
random.random()
```

```
Functions  4-21

makeFish with parameters

In lecture, you will modify the makeFish function definition and invocations to produce the fishtank picture shown below.

```python
# Zero-Parameter Functions

Sometimes it's helpful to define/use functions that have zero parameters. Note: you still need parentheses after the function name when defining and invoking the function.

```python
def rocks():
    print('CS111 rocks!')
def rocks3():
    rocks()
    rocks()
    rocks()
```

Python libraries have useful built-in functions with zero parameters and a return value:

```python
import random
random.random()
```

```
Functions  4-21

fish_with_functions.py

This makes a new fish Layer via a function call rather than a clone. With parameters (see next few slides), functions are more powerful than clones.

```python
def makeFish():
    fish = Layer()  # fish layer
    # body of the fish
    body = Ellipse(100,50,Point(0,0))
    body.setFillColor('yellow')
    fish.add(body)
    # green tail of the fish
    tail = Polygon()
    tail.addPoint(Point(-50,0))
    tail.addPoint(Point(-75,25))
    tail.addPoint(Point(-75,-25))
    tail.setFillColor('green')
    fish.add(tail)
    # black eye of the fish
    eye = Circle(5,Point(25,-5))
    eye.setFillColor('black')
    fish.add(eye)
return fish
```

```
Functions  4-22

Functions

In lecture, you will modify the makeFish function definition and invocations to produce the fishtank picture shown below.

```python
# Zero-Parameter Functions

Sometimes it's helpful to define/use functions that have zero parameters. Note: you still need parentheses after the function name when defining and invoking the function.

```python
def rocks():
    print('CS111 rocks!')
def rocks3():
    rocks()
    rocks()
    rocks()
```

Python libraries have useful built-in functions with zero parameters and a return value:

```python
import random
random.random()
```

```
Functions  4-22

Functions

In lecture, you will modify the makeFish function definition and invocations to produce the fishtank picture shown below.

```python
# Zero-Parameter Functions

Sometimes it's helpful to define/use functions that have zero parameters. Note: you still need parentheses after the function name when defining and invoking the function.

```python
def rocks():
    print('CS111 rocks!')
def rocks3():
    rocks()
    rocks()
    rocks()
```

Python libraries have useful built-in functions with zero parameters and a return value:

```python
import random
random.random()
```

```
Functions  4-23

Zero-Parameter Functions

Sometimes it's helpful to define/use functions that have zero parameters. Note: you still need parentheses after the function name when defining and invoking the function.

```python
def rocks():
    print('CS111 rocks!')
def rocks3():
    rocks()
    rocks()
    rocks()
```

Python libraries have useful built-in functions with zero parameters and a return value:

```python
import random
random.random()
```

```
Functions  4-23

Zero-Parameter Functions

Sometimes it's helpful to define/use functions that have zero parameters. Note: you still need parentheses after the function name when defining and invoking the function.

```python
def rocks():
    print('CS111 rocks!')
def rocks3():
    rocks()
    rocks()
    rocks()
```

Python libraries have useful built-in functions with zero parameters and a return value:

```python
import random
random.random()
```

```
Functions  4-24

Zero-Parameter Functions

Sometimes it's helpful to define/use functions that have zero parameters. Note: you still need parentheses after the function name when defining and invoking the function.

```python
def rocks():
    print('CS111 rocks!')
def rocks3():
    rocks()
    rocks()
    rocks()
```

Python libraries have useful built-in functions with zero parameters and a return value:

```python
import random
random.random()
```
Updated Function diagrams

max(), min()
square()
hypotenuse()

random.random()
makeFish()

print(), help()
verse()
printBanner()
printSquare()

raw_input(),
squarePrintArg()

Functions  4-25

Unindented function body

Python is unusual among programming languages in that it uses indentation to determine what’s in the body of a function.

def square(x):
    return x**x

You can indent by using the TAB character in the keyboard. Alternatively, you can use a consistent number of spaces (e.g. 4).

The following definition is *incorrect* because the body isn’t indented:

def square(x):
    return x**x

In general, when the indentation is wrong, you’ll see error messages that point you to the problem, e.g.:

IndentationError: expected \t in indented block
SyntaxError: 'return' outside function

Visualizing Code Execution with the Python Tutor

Python Tutor: [http://www.pythontutor.com/visualize.html](http://www.pythontutor.com/visualize.html)

It automatically shows many (but not all) aspects of our CS111 Python function call model. You’ll use it in Lab.

The None value and NoneType

- Python has a special None value (of type NoneType), which Python normally doesn’t print.
- A function without an explicit return statement actually returns the None value!

```
In [2]: None
Out[2]: None

In [3]: type(None)
Out[3]: NoneType

In [4]: None + None
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-28a1675638b9> in <module>()
----> 1 None + None

TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```

On slide 4-12, this is the real reason that the expression
print_square(3) + print_square(4) causes an error.
**Fruitful vs. None Functions**

We will call functions that return the `None` value `None functions*`. None functions are invoked to perform an action (e.g. print characters, change object state), not to return a result.

We will call functions that return a value other than `None` are **fruitful functions**. Fruitful functions return a meaningful value. Additionally, they may also perform an action.

* In Java, methods that don't return a value are `void` methods. So we may sometimes use "void functions" as a synonym for "None functions"

**Functions w/ local variables: hypotenuse2 [1]**

```python
def hypotenuse2(a, b):
    sqa = square(a)
    sqb = square(b)
    sqsum = sqa + sqb
    return math.sqrt(sqsum)
```

**Functions w/ local variables: hypotenuse2 [2]**

```python
def hypotenuse2(a, b):
    sqa = square(a)
    sqb = square(b)
    sqsum = sqa + sqb
    return math.sqrt(sqsum)
```

**Local variables in the Frame Model**

We’ve seen numerous examples of functions that use local variables, but we haven’t explained how local variables work in the execution model with function frames.

We’ll do that now with the `hypotenuse2` function:
Test your knowledge

1. What is the difference between a function definition and a function call?
2. What is the difference between a parameter and an argument? In what context is each of them used?
3. Is it OK to use the same parameter names in more than one function definition? Why or why not?
4. Suppose the parameters of the hypotenuse function in 4-15 are renamed from a and b to side1 and side2. Does the function still work as expected? Does any other part of the program “know” that the parameter names have been changed?
5. Can a function have a return value and no side effects? Side effects and no return value? Both side effects and a return value?
6. Can a function whose definition lacks a return statement be called within an expression?
7. What would happen if we swap the order of print and return in the definition of squarePrintArg in slide 4-11. Why? If you cannot imagine it, test it out in Canopy.
8. What is the value of using the function call model?
9. What is indentation and where it is used within Python?