Abstracting with Functions

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
Then if we want every fish to have a hat, we just modify our one prototype fish before we clone it.

# 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)
```

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 usually take inputs and return outputs based on those inputs

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

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>max(7, 3)</code></td>
<td>7</td>
</tr>
<tr>
<td><code>min(7, 3, 2, 9)</code></td>
<td>2</td>
</tr>
<tr>
<td><code>type(123)</code></td>
<td>int</td>
</tr>
<tr>
<td><code>len('CS111')</code></td>
<td>5</td>
</tr>
<tr>
<td><code>str(4.0)</code></td>
<td>'4.0'</td>
</tr>
<tr>
<td><code>int(-2.978)</code></td>
<td>-2</td>
</tr>
<tr>
<td><code>float(42)</code></td>
<td>42.0</td>
</tr>
<tr>
<td><code>round(2.718, 1)</code></td>
<td>2.7</td>
</tr>
<tr>
<td><code>raw_input('Enter your age: ')</code></td>
<td>'19'</td>
</tr>
</tbody>
</table>
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
```

Function diagrams summarize what functions do

Anatomy of a User-defined Function

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

### Function definitions

- **Definition**
  - `def square(x):`
  - `return x * x`
  - **Parameter**
  - **Header**
  - **Body**
  - **Keyword indicating return value**

### Function calls/invocations

- **Calls**
  - `square(5)` ➔ **25**
  - `square(10)` ➔ **100**
  - `square(-3)` ➔ **9**

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
def square(a):
    return a * a

def square(x):
    return x * x

def square(num):
    return num * num

def square(a_long_parameter_name):
    return a_long_parameter_name * a_long_parameter_name
```
We need a model to understand how function calls work.

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

A function call is “replaced” by its returned value

```
17 + square(2 + 3)
```

Step 1: evaluate all argument expressions to values (e.g., numbers, strings, objects ...)

Step 2: 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.

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?)

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

```
17 + square(5)
```

```
17 + square(5)
```

```
17 + 25
```

```
42
```

Unindented function body

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

```python
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:

```python
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 an indented block`
- `IndentationError: unindent does not match any outer indentation level`

Multiple parameters, calling other functions

Functions may take multiple parameters/arguments, separated by commas. They may also 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
```

Note the use of Python’s `math` module.
Function call model for hypotenuse(3,4) [1]

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

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

```
return math.sqrt(square(3) + square(b))
```

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

```
return math.sqrt(square(3) + square(4))
```

Visualizing Code Execution with the Python Tutor

Python Tutor: 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.
**Exercise: Define average**

Define a function named `average` that takes two numbers and returns their average.

Use it (1) to compute the average of 6 and 10 and (2) to compute the average of 5 and 10.

Does it work as you expect? Why or why not?

---

**Local variables**

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 cannot be referred outside the function too.

---

**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')
printBanner('Pied Piper')
```

---

**How many days is a million seconds?**

```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
```
Old MacDonald had a farm, EE-I-EE-I-O,
And on that farm he had a cow, EE-I-EE-I-O,
With a moo moo here and a moo moo there
Here a moo, there a moo, everyone a moo moo
Old MacDonald had a farm, EE-I-EE-I-O.

Old MacDonald had a farm, EE-I-EE-I-O,
And on that farm he had a chicken, EE-I-EE-I-O,
With a cluck cluck here and a cluck cluck there
Here a cluck, there a cluck, everyone a cluck cluck
Old MacDonald had a farm, EE-I-EE-I-O.

Old MacDonald had a farm, EE-I-EE-I-O,
And on that farm he had a horse, EE-I-EE-I-O,
With a neigh neigh here and a neigh neigh there
Here a neigh, there a neigh, everyone a neigh neigh
Old MacDonald had a farm, EE-I-EE-I-O.

Old MacDonald had a farm, EE-I-EE-I-O,
And on that farm he had a sheep, EE-I-EE-I-O,
With a baa baa here and a baa baa there
Here a baa, there a baa, everyone a baa baa
Old MacDonald had a farm, EE-I-EE-I-O.

Functions capture common patterns

```python
def verse(animal, noise):
    print('Old MacDonald had a farm, EE-I-EE-I-O,')
    print('And on that farm he had a ' + animal + ', EE-I-EE-I-O,')
    # Exercise: complete this function body in notebook!
    print('With a ' + noise + ' ' + noise + ' here and a ' + noise + ' ' + noise + ' there')
    print('Here a ' + noise + ', there a ' + noise + ', everyone a ' + noise + ' ' + noise)
    print('Old MacDonald had a farm, EE-I-EE-I-O.

verse('cow', 'moo')
verse('chicken', 'cluck')
verse('horse', 'neigh')
verse('sheep', 'baa')
```

return vs. print: squaring example [1]

- **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
```

```python
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
```

return vs. print: squaring example [2]

```python
def printSquare(a):
    print('The argument of square is ' + str(a) + ' is ' + str(square(a)))
```

```python
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)

printSquare() does not return a number, so it doesn't make sense to add the two invocations!
```
Function Abstraction: Fishtank Revisited

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

```
# 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)
```

Functions 3-25

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.

```
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 3-26

```
# 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)
```

Functions 3-26

makeFish with parameters

In the `fish_with_functions.py` file, modify the `makeFish` function definition and invocations to produce the fishtank picture shown below.

```
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 3-26

Incremental Development [1]

Step 1: modify the `makeFish` function definition and invocations to produce fish that have a different body color, as shown below.
Incremental Development [2]

**Step 2:** modify the `makeFish` from Step 1 to now add the tail color and perform the right invocations to generate:

![Image of fish with multicolored tails]

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()
```

Out [...]

A random float value between 0 and 1.

Incremental Development [3]

**Step 3:** modify the `makeFish` from Step 2 to now add the eye radius.

What other hard-coded values from the original `makeFish` function can you replace with parameters, so that the `makeFish` function becomes a prototypical fish?

You can also add some code to generate a fin, and modify the function definition and invocation accordingly.

Function diagrams
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:

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

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)
```

Functions w/local variables: `hypotenuse2` [3]

```python
def hypotenuse2(a, b):
    sqa = square(a)
    sqb = square(b)
    sqsum = sqa + sqb
    return math.sqrt(sqsum)
```
functions w/local variables: hypotenuse2 [4]  

```
a 3 b 4 sqa 9 sqb 16
sqa = 9
sqb = 16
sqsum = 25
return math.sqrt(sqsum)
```

```
a 3 b 4 sqa 9 sqb 16
sqsum 25
sqa = 9
sqb = 16
sqsum = 25
return math.sqrt(sqsum)
```

```
a 3 b 4 sqa 9 sqb 16
sqsum 25
sqa = 9
sqb = 16
sqsum = 25
return math.sqrt(25)
```

```
a 3 b 4 sqa 9 sqb 16
sqsum 25
sqa = 9
sqb = 16
sqsum = 25
return 5.0
```

---

**The None value and NoneType**

- Python has 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: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```

On slide 3-26, 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.

```
square  
square_print  
hypotenuse  
hypotenuse2
```

```
def f1():  
x1 = 7  
print(x1)
```

```
def f2():  
x2 = 7  
print(x2)
```

```
def f3():  
global x3  
x3 = 7  
print(x3)
```

```
def f4():  
x4 = 7  
print(x4)
```

```
x1 = 5  
print(x1)
```

```
x2 = 5  
print(x2)
```

```
x3 = 7  
print(x3)
```

```
x4 = 7  
print(x4)
```

```
gen  
7  
NameError: name 'x4' is not defined
```

Variables assigned outside any function are known as **global variables**.

If a variable is assigned within a function definition, it is assumed to be a **local variable** of the function unless the `global` declaration is used to indicate it references a global variable outside the function instead.

---

**Global Variables / Global vs Local Scope**

* 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”.

---

---
Python Tutor with globals and locals

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

Counter function

How can we define a zero-parameter count function that returns the number of times it has been called?

```python
c = 0  # global variable storing the current count
def count():
    global c  # What happens if we forget this?
    c = c + 1
    return c
```

- count() → 1
- count() → 2
- count() → 3
- count() → 4

```python
def cube(x):
    result = x**x
    return result

side = 10
print(cube(side))
```