MORE ON FUNCTION OBJECTS

LECTURE 08–2

JIM FIX, REED COLLEGE CSCI 121

COURSE INFO

- **THIS FRIDAY:** project 2 ***ciphers*** is due… any questions?
- **Midterm Exam** results and solutions…
- **Project 3** on object orientation (bird simulation) will be posted
- **Today:**
  - Functions as data objects (a.k.a. "higher order functions") ***cont'd***
    - ➡ Expressing functions succinctly using `lambda`.
    - ➡ Returning functions as values.
  - Python Tutor demos of function that return functions.
  - I will post a (very short) **Homework 8**; only three exercises; due???

THE HIGHER-ORDER FUNCTION FEATURES OF PYTHON

How?

- Python treats function as objects. This gives Python certain nifty features.

Generally:

Languages that have ***higher-order function features*** allow you to:

- Pass functions/procedures as arguments to other functions/procedures.
- Express functions succinctly and anonymously (using `lambda`).
- Assign variables to be function objects, ***and***
- Return functions back from other functions.

THE HIGHER-ORDER FUNCTION FEATURES OF PYTHON

How?

- Python treats function as objects. This gives Python certain nifty features.

Generally:

Languages that have ***higher-order function features*** allow you to:

- Pass functions/procedures as arguments to other functions/procedures.
- Express functions succinctly and anonymously (using `lambda`).
- Assign variables to be function objects, ***and***
- Return functions back from other functions.

RECALL: LAMBDA SYNTAX

The `lambda` construct allows you to express a function without naming it.

- ➡ It provides *anonymous function definition*

Here is the syntax:

`lambda` ***parameters*** `:` ***expression for computed value***

- It constructs a function object that returns the computed value described.

Some other examples, named using variable assignment:

```
squarer = (lambda x: x * y)
adder = (lambda x: x + y)
successor = lambda number: number + 1
sum_squares = lambda x,y : x*x + y*y
apply_twice = lambda f,x : f(f(x))
say_hi = lambda n: print(“hi!”*n)
echo = lambda : print(input(“?”)+”!”)
```

LAMBDAS THAT RETURN FUNCTION OBJECTS

- These two each create a function with their parameter:

```
>>> make_adder = (lambda x: (lambda y: x + y))
>>> make_repeater = (lambda s: (lambda n: print(s * n)))
>>> (make_adder(3))(4)
7
>>> adds_three = make_adder(3)
>>> adds_three(4)
7
>>> hello_repeater = make_repeater(“hello”)
>>> hello_repeater(3)
hellohellohello
>>> hello_repeater(10)
hellohellohellohellohellohellohellohellohellohello
```

SOME LIVE EXERCISES

- **Exercise 1:** Using only `lambda`, write a function `twoDigits` that does this:

```
>>> twoDigits = lambda ????
>>> twoDigits(3,7)
37
>>> twoDigits(8,9)
89
```

- The code below builds a quadratic function object, then uses it:

```
>>> q = lambda x: 5*x**2 + 3*x - 1
>>> q(3)
53
>>> q(-1)
1
```

- **Exercise 2:** write a function `makeQuadratic` that behaves like this:

```
>>> q = makeQuadratic(5,3,-1)
>>> q(3)
53
>>> q(-1)
1
```

- ***BONUS* Exercise 3:** write a function `makeAppender` that behaves like this:

```
>>> xs = [1,2,3]
>>> more_xs = makeAppender(xs)
>>> more_xs(10)
>>> more_xs(100)
>>> xs
[1, 2, 3, 10, 100]
```

EXERCISE ANSWERS

- Here are possible answers:

```
twoDigits = lambda tens, ones: 10 * tens + ones
makeQuadratic = lambda a, b, c: (lambda x: a*x**2 + b*x + c)
makeAppender = lambda xs: (lambda x: xs.append(x))
```

RETURNING FUNCTIONS

- Here is a way of writing **`makeQuadratic`**:

```
def makeQuadratic(a,b,c):
    return (lambda x: a*x**2 + b*x + c)
```

RETURNING FUNCTIONS

- Here is a way of writing **`makeQuadratic`**:

```
def makeQuadratic(a,b,c):
    return (lambda x: a*x**2 + b*x + c)
```

- We can then do this:

```
>>> q = makeQuadratic(5,3,-1)
>>> q(3)
53
>>> q(-1)
1
```

RETURNING FUNCTIONS

- Here is a way of writing **`makeQuadratic`**:

```
def makeQuadratic(a,b,c):
    return (lambda x: a*x**2 + b*x + c)
```

- We can then do this:

```
>>> q = makeQuadratic(5,3,-1)
>>> q(3)
53
>>> q(-1)
1
```

- And we can also do this:

```
>>> r = makeQuadratic(1,0,-1)
>>> r(3)
8
>>> r(-1)
0
```

RETURNING FUNCTIONS

- Here is a way of writing **`makeQuadratic`**:

```
def makeQuadratic(a,b,c):
    return (lambda x: a*x**2 + b*x + c)
```

- We can then do this:

```
>>> q = makeQuadratic(5,3,-1)
>>> q(3)
53
>>> q(-1)
1
```

- And we can also do this:

```
>>> r = makeQuadratic(1,0,-1)
>>> r(3)
8
>>> r(-1)
0
```

- The function **`makeQuadratic`** is a kind of ***function factory***.

WHY THIS WORKS

- Here is a script in Python Tutor: https://tinyurl.com/5abcyuv2

```
def makeQuadratic(a,b,c):
    return (lambda x: a*x**2 + b*x + c)

q = makeQuadratic(5,3,-1)
r = makeQuadratic(1,0,-1)
print(q(3))
print(r(3))
```

WHY THIS WORKS

- Here is a script in Python Tutor:

```
def makeQuadratic(a,b,c):
    return (lambda x: a*x**2 + b*x + c)

q = makeQuadratic(5,3,-1)
r = makeQuadratic(1,0,-1)
print(q(3))
print(r(3))
```

- Each function has a **parent frame** giving the context when it was invented.
  - ➡ These are kept with the function object.
  - ➡ For the two lambdas, each will know its own `a`, `b`, and `c`.

AN ADDER FUNCTION FACTORY

- Let's define a function that produces adding functions:

```
def makeAdder(by_this_much):
    return (lambda x: x + by_this_much)
```

- Here it is in use:

```
>>> successor = makeAdder(1)
>>> by_ten = makeAdder(10)
>>> successor(7)
8
>>> successor(70)
71
>>> by_ten(7)
17
>>> by_ten(70)
80
>>> (makeAdder(100))(7)
107
```

ALTERNATIVES FOR WRITING AN ADDER-MAKER

- Here are several different ways of writing the code for **`makeAdder`**:

```
def makeAdder(by_this_much):
    return (lambda x: x + by_this_much)

def makeAdder(by_this_much):
    adder = (lambda x: x + by_this_much)
    return adder

def makeAdder(by_this_much):
    def adder(x):
        return x + by_this_much
    return adder

makeAdder = (lambda btm: (lambda x: x + btm))
```

- We see that **`def`** is just a multi-line assignment statement for functions.

A PROCEDURE FACTORY

```
def makeLineRepeater(some_text):
    def line_repeater(number):
        i = 0
        while i < number:
            print(some_text)
            i = i+1
    return line_repeater
```

```
>>> greeter = makeRepeater(“hello”)
>>> ouchie = makeRepeater(“ow!”)
>>> greeter(3)
hello
hello
hello
>>> ouchie(5)
ow!
ow!
ow!
ow!
ow!
```

AN INPUT PROCEDURE FACTORY

```
def makeRequester(prompt, conversion, condition):
    def requester():
        while True:
            entry = input(prompt)
            value = conversion(entry)
            if condition(value):
                return value
            print("Not what we requested.")
    return requester

good_area = lambda x: x >= 0
area_get = makeRequester(“Enter an area: ", float, good_area)
ok_ans = lambda s: s == "yes" or s == "no"
answer_get = makeRequester(“yes / no? ", lambda x:x, ok_ans)
is_die = lambda d: (d >= 1) and (d <= 6)
roll_get = makeGetter("What did you roll? ", int, is_die)
```

A TEMPLATE FOR FUNCTION FACTORIES

```
def function_factory(which-one-you-want...):
    def some_function(x1,x2,…):
        # Describe how it acts on x1, x2, etc
        # according to which-one-you-want...
        …
        return …
    return some_function
```

- Here is its application for **`makeAdder`**:

```
def makeAdder(dx):
    def adder(x):
        return x+dx
    return adder
```

ANOTHER TEMPLATE FOR FUNCTION FACTORIES

```
def function_factory(which-one-you-want…):
    some_function = (lambda x1,x2,…: … )
    return some_function
```

- Here is its application for **`makeAdder`**:

```
def makeAdder(dx):
    adder = (lambda x: x+dx)
    return adder
```

YET ANOTHER TEMPLATE FOR FUNCTION FACTORIES

```
def function_factory(which-one-you-want…):
    return (lambda x1,x2,…: … )
```

- Here is its application for **`makeAdder`**:

```
def makeAdder(dx):
    return (lambda x: x+dx)
```

AND YET ANOTHER TEMPLATE FOR FUNCTION FACTORIES

```
function_factory = (lambda which-one-you-want...: (lambda x1,x2,…: … ))
```

- Here is its application for **`makeAdder`**:

```
makeAdder = (lambda dx: (lambda x: x+dx))
```

FRAME DEMO

- Execute this script in Python Tutor at this link: https://tinyurl.com/3tw67f5p

```
def make_adder(dx):
    def adder(x):
        return x + dx

f = make_adder(7)
g = make_adder(8)

print(f(11))
print(g(12))
```

- Pay attention to the frames active each time **`make_adder`** executes.
  - ➡ Two different frames are active when each adder object is constructed.
  - ➡ Each will be the context frame for its adder.
  - ➡ One has **`dx`** of 7, the other has a **`dx`** of 8.

FRAME DEMO

- Execute this script in Python Tutor at this link: https://tinyurl.com/3tw67f5p

```
def make_adder(dx):
    def adder(x):
        return x + dx

f = make_adder(7)
g = make_adder(8)
print(f(11))
print(g(12))
```

- When each adder is called…
  - A new frame is created and activated with each call.
    - ➡ One with `x` of 11, one with `x` of 12.
  - Its parent frame is the context frame of its adder function.
    - ➡ One has a `dx` of 7, the other has a `dx` of 8.

FRAME DEMO

- Execute this script in Python Tutor at this link: https://tinyurl.com/3tw67f5p

```
def make_adder(dx):
    def adder(x):
        return x + dx

f = make_adder(7)
g = make_adder(8)

print(f(11))
print(g(12))
```

- *A chain of parent frames is followed until a variable name is found.*

PARENT FRAMES

- Execution of this script:

```
def make_adder(dx):
    def adder(x):
        return x + dx
    return adder

f = make_adder(7)
g = make_adder(8)

print(f(11))
print(g(12))
```