MORE ON FUNCTION OBJECTS

LECTURE 07–2

JIM FIX, REED COLLEGE CSCI 121

COURSE INFO

- **Today:** functions as data objects (a.k.a. "higher order functions") cont'd
  - expressing functions succinctly using `lambda`
  - returning functions as values
- **Reading:**
  - PP Chapter 2.1
  - CP Chapter *1.6*
- **THIS MONDAY:** midterm exam on HWs 1-5

HIGHER ORDER FUNCTIONS

- Python treats functions as objects.
  - This means we can hand functions to other functions.
    - Functions can be passed as parameters.

- Functions that take functions as parameters are ***higher order functions.***

- Such functions are "reasoning about" the functions they are given.

THE HIGHER-ORDER FUNCTION FEATURES OF PYTHON

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

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.

EXAMPLE: FINDING A MINIMUM VALUE

- **Given:** the polynomial $p(x) = x^4 - 8x^3 + 6x - 4$
- **Find:** which integer from 3 to 10 yields the lowest value?

EXAMPLE: FINDING A MINIMUM VALUE

The code below generalizes on the *function being checked*:

```
def p(x):
    return x**4 – 8*x**3 + 6*x – 4

def argument_for_min(some_function,start,end):
    min_so_far = some_function(start)
    where_seen = start
    i = start + 1
    while i <= end:
        if p(i) < min_so_far:
            min_so_far = some_function(i)
            where_seen = i
        i = i + 1
    return where_seen

print(argument_for_min(p,3,10))
print(argument_for_min(p,-20,5))
print(argument_for_min(p,387,501))
```

EXAMPLE: USING IT FOR TWO DIFFERENT FUNCTIONS!

```
def argument_for_min(some_function,start,end):
    min_so_far = some_function(start)
    where_seen = start
    i = start + 1
    while i <= end:
        if p(i) < min_so_far:
            min_so_far = some_function(i)
            where_seen = i
        i = i + 1
    return where_seen

def p(x):
    return x**4 – 8*x**3 + 6*x – 4

def another(arg):
    return 3*arg**5 – 100*arg**2 + 99

print(argument_for_min(p,3,10))
print(argument_for_min(another,3,10))
```

NEEDING DEF CAN SEEM WORDY...

```
def argument_for_min(some_function,start,end):
    min_so_far = some_function(start)
    where_seen = start
    i = start + 1
    while i <= end:
        if p(i) < min_so_far:
            min_so_far = some_function(i)
            where_seen = i
        i = i + 1
    return where_seen

def f1(x):
    return x * x - 3
def f2(x):
    return x - 3 * abs(x)
def f3(x):
    return x ** 2 - 1

print(argument_of_min(f1,-5,3))
print(argument_of_min(f2,-5,3))
print(argument_of_min(f3,-5,3))
```

HIGHER-ORDER FUNCTION FEATURES

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.

HIGHER-ORDER FUNCTION FEATURES

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.

CAN USE LAMBDA EXPRESSIONS INSTEAD

```
def argument_for_min(some_function,start,end):
    min_so_far = some_function(start)
    where_seen = start
    i = start + 1
    while i <= end:
        if p(i) < min_so_far:
            min_so_far = some_function(i)
            where_seen = i
        i = i + 1
    return where_seen


print(argument_of_min(lambda x: x * x - 3,-5,3))
print(argument_of_min(lambda x: x - 3 * abs(x),-5,3))
print(argument_of_min(lambda x: x ** 2 - 1,-5,3))
```

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:

```
square = lambda a: a * a
successor = lambda number: number + 1
sum_squares = lambda x,y : x*x + y*y
two_digit = lambda d1,d2 : 10*d1 + d2
apply_twice = lambda f,x : f(f(x))
say_hi = lambda n: print(“hi!”*n)
echo = lambda : print(input(“?”)+”!”)
```

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***

HISTORY

The notation comes from logician A. Church who explored schemes for formalizing logic, computation, and proof (1930s).

λ f. λ f. f (f x) would be his notation for "apply twice."

λ n. λ f. λ x. f (n f x) was his successor function for "Church numerals."

λ f. (λ x. f (x x)) (λ x. f (x x)) was how he could express recursion.

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***

HISTORY

J. McCarthy adopted it for his symbolic reasoning language **LisP** (1950s).

`(lambda (x) (+ x 1))` is how you write the successor function in LisP.

`(lambda (d1 d2) (+ (* d1 10) d2))` is how you write two digit.

`(lambda (f x) (f (f x)))` might be how you write apply twice.

`(lambda (f) (lambda (x) (f (f x)))` might also be how (using "Currying").

RETURNING FUNCTION OBJECTS?

The lambda notation feels powerful.

- 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
```

- Can we do this?

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

- If we can, then we could do this:

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

HIGHER-ORDER FUNCTION FEATURES

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.

HIGHER-ORDER FUNCTION FEATURES

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.***

RETURNING FUNCTIONS

- If we write this higher-order function:

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

- Then we can 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***.

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

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

ANOTHER PROCEDURE-MAKER

```
def tablePrinterFor(op):

    def printTable(rows,cols):
        for i in range(rows):
            for j in range(cols):
                value = op(i,j)
                print(value,end='\t')
            print()

    return printTable


>>> from operator import mul
>>> mult_table = tablePrinterFor(mul)
>>> mult_table(4,6)    # Prints a 4x6 mult. table.
…
>>> mult_table(12,12) # Prints a 12x12 table.
…
```

AN INPUT PROCEDURE FACTORY

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

good_area = lambda x: x >= 0
area_get = makeGetter(“Enter an area: ", float, good_area)
ok_ans = lambda s: s == "yes" or s == "no"
answer_get = makeGetter("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:

```
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))
```

FRAME DEMO #2

- Execute this script in Python Tutor at this link: https://tinyurl.com/yc7um43e

```
five = 5
def add_five(x):
    return x + five

print(add_five(10))
five = five + 1
print(add_five(10))
```

- Pay attention to how execution of **`add_five`** accesses the global frame.
  - The global frame is active when **`add_five`** is defined.
  - It is the **`add_five`** function's "*context frame*."
  - It becomes the ***parent frame*** of the frame that activates when **`add_five`** is called.

FRAME DEMO #3

- Execute this script in Python Tutor at this link: https://tinyurl.com/4ycfbcpv

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

successor = make_adder(1)
by_ten = make_adder(10)
print(successor(7))
print(by_ten(9))
```

- Pay attention to the frames active each time make_adder executes.
  - ➡ Two different frames are active when each adder object is constructed.
  - ➡ One has `dx` of 1, the other has a `dx` of 10.
  - ➡ Thus each adder's function object gets its own context frame.

FRAME DEMO #3

- Execute this script in Python Tutor at this link: https://tinyurl.com/4ycfbcpv

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

successor = make_adder(1)
by_ten = make_adder(10)
print(successor(7))
print(by_ten(9))
```

- When each adder is called…
  - A new frame is created and activated with the call.
    - ➡ One with x of 7, one with x of 9.
  - Its parent frame is the context frame of its adder function.
    - ➡ One has a dx of 1, the other has a dx of 10.

FRAME DEMO #3

- Execute this script in Python Tutor at this link: https://tinyurl.com/4ycfbcpv

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

successor = make_adder(1)
by_ten = make_adder(10)
print(successor(7))
print(by_ten(9))
```

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

FRAME ANIMATION...

- 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))
```