FUNCTION OBJECTS

LECTURE 09-1

JIM FIX, REED COLLEGE CSCI 121

COURSE INFO

- **Project 3** will be posted today. It's due Friday, April 14th.
  - **hawk-dove**: a simulation of evolving birds that compete for resources

- **Today:** functions as data objects, a.k.a. "higher order functions"
  - expressing functions succinctly using **`lambda`**
  - passing functions as arguments
  - returning functions as values

- **Reading:** CP Chapter ***1.6 Higher-Order Functions***
- **NEXT MONDAY:** a quiz on recursion

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

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

Here is a script that computes that minimum:

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

min_so_far = p(3)
where_seen = 3
i = 4
while i <= 10:
    if p(i) < min_so_far:
        min_so_far = p(i)
        where_seen = i
    i = i + 1
print(where_seen)
```

A TEMPLATE FOR FINDING MINIMUMS

Note that there is a **template** for performing this algorithm. Can work for...

- *...any function*
- *...any start value*
- *...any end value*

```
min_so_far = some_function(3)
where_seen = start
i = start + 1
while i <= end:
    if some_function(i) < min_so_far:
        min_so_far = some_function(i)
        where_seen = i
    i = i + 1
print(where_seen)
```

EXAMPLE: FINDING A MINIMUM VALUE

The code below generalizes on the range we check:

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

def argument_for_min_p(start,end):
    min_so_far = p(start)
    where_seen = start
    i = start + 1
    while i <= end:
        if p(i) < min_so_far:
            min_so_far = p(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: FINDING A MINIMUM VALUE

The code below also 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))
```

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.

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))
```

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 examples, named using variable assignment:

```
square = lambda a: a * a
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 : print("hi!")
```

A HIGHER-ORDER PROCEDURE

How about this procedure?

```
def sequence_report(name, seq, n):
    ????
```

Here is how I'd like it to work:

```
>>> sequence_report("fib",fibonacci,9)

 n | fib(n)
---+--------
 1 | 1
 2 | 1
 3 | 2
 4 | 3
 5 | 5
 6 | 8
 7 | 13
 8 | 21
 9 | 34
```

A SEQUENCE REPORTER

Here is the code for it:

```
def sequence_report(name, seq, n):
  print(" n | " + name + "(n)")
  print("-"*3 + "+" + "-"*(len(name)+5))
  i = 1
  while i <= n:
      print(" "+str(i)+" | "+str(seq(i)))
      i = i + 1
```

YET ANOTHER HIGHER-ORDER PROCEDURE

Q: What does this procedure do?

A: ?

```
def abcde(op,size):
  i = 1
  while i <= size:
      j = 1
      while j <= size:
          value = op(i,j)
          print(str(value),end='\t')
          j = j + 1
      print()
      i = i + 1
```

A MULTIPLICATION TABLE

This is what it does:

```
>>> mul = lambda x,y: x * y
>>> abcde(mul,5)
1   2   3   4   5
2   4   6   8   10
3   6   9   12  15
4   8   12  16  20
5   10  15  20  25
```

A MULTIPLICATION TABLE

This is what it does:

```
>>> from operator import mul
>>> abcde(mul,5)
1   2   3   4   5
2   4   6   8   10
3   6   9   12  15
4   8   12  16  20
5   10  15  20  25
```

YET ANOTHER HIGHER-ORDER PROCEDURE

Q: What does this procedure do?

A: It produces a table for any two-parameter function `op`.

```
def table(op,size):
  i = 1
  while i <= size:
      j = 1
      while j <= size:
          value = op(i,j)
          print(str(value),end='\t')
          j = j + 1
      print()
      i = i + 1
```

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))
```