PYTHON PROCEDURES; PASSING FUNCTIONS TO FUNCTIONS

LECTURE 03-2

PROCEDURES
FUNCTION OBJECTS AS PARAMETERS
PROJECT 1

JIM FIX, REED COLLEGE CSCI 121

PROGRAMMER-DEFINED PROCEDURES

- Python has the same **`def`** syntax for defining ***procedures***
  - ➡ This is my term for a "function that does not return a value."
  - ➡ Instead, it does some stuff, performs some actions.
- For example

```
def printBoxTop(size):
    dashes = "-" * size
    print("+" + dashes + "+")

def printBox(width):
    printBoxTop(width)
    print("|" + (" "*width) + "|")
    printBoxTop(width)
```

- Below is its use. It's as if we've invented a **`printBox`** statement.

```
>>> printBox(4)
+----+
|    |
+----+
>>>
```

SYNTAX: PROCEDURE DEFINITION

Below gives a template for procedure definitions:

```
def procedure-name(parameter-list):
    lines of statements that compute using the parameters
    ...
    return
```

- The last line is often a `return` statement, but it isn't needed.
- There can also be `return` statements within the code.
  - ➡ These lead Python to exit the procedure as soon as they are reached.
  - ➡ Control returns back to where the procedure was called, continues there.

EXAMPLE SCRIPT WITH PROCEDURES

```
def printBoxTop(size):
    dashes = "-" * size
    print("+" + dashes + "+")

def greetTheUser(name):
    print("Hi, " + name + ". Nice to meet ya!")
```

```
def printBox(w):
    printBoxTop(w)
    print("|" + (" " * w) + "|")
    printBoxTop(w)

user = input("What's your name? ")
greetTheUser(user)
print("I'd like to make you a box.")
width = int(input("How wide of a box would you like? "))
printBox(width)
print("Here is one that is twice as wide:")
printBox(width * 2)
```

PROCEDURES RETURN THE NONE VALUE

- All three of these procedures do the exact same thing:

```
def greetThenReturn_version1(name):
    print("Hi, " + name + ".")

def greetThenReturn_version2(name):
    print("Hi, " + name + ".")
    return
```

```
def greetThenReturn_version3(name):
    print("Hi, " + name + ".")
    return None
```

- The first implicitly returns `None`. The first explictly returns but implictly returns `None`. The third explicitly returns the `None` value.

NONE IS WEIRDLY HANDLED BY THE PYTHON INTERPRETER

- Here is some fun with `None`, and with procedures (that return `None`):

```
>>> print("hello")
hello
>>> print(None)
None
>>> "hello"
'hello'
>>> None
>>> 3+4
7
>>> print(print("hello"))
hello
None
>>> greetThenReturn("Jim")
Hello, Jim.
>>> print(greetThenReturn("Jim"))
Hello, Jim.
None
```

FUNCTIONS VS. PROCEDURES

- "**Function**":
  - A function gets passed some parameters, executes, and then returns a result.
  - A function is used within an expression.

- "**Procedure**":
  - A procedure is something that (typically) performs some action/work but does not return a value.
  - A procedure is used as a statement of a line of code.
  - When a procedure's work is done, Python continues executing after the line where it was called. (Control "jumps" then returns.)

FUNCTIONS VS. PROCEDURES (CONT'D)

- In Python, procedures are really just functions.
  - Python doesn't distinguish procedures from functions.
  - This is just my personal dichotomy, from older languages (Pascal, C).

➡ Functions can perform actions (print, get input) too, before they return.

PASSING FUNCTIONS TO FUNCTIONS

LECTURE 03–2
INTRO TO HIGHER ORDER FUNCTIONS

JIM FIX, REED COLLEGE CSCI 121

FUNCTION OBJECTS

We've seen some evidence that Python treats functions like data

```
>>> def square(x):
...     return x * x
...
>>> square
<function square at 0x104cbb7e0>
>>> type(square)
<class 'function'>
>>> def abs(x):
...     if x < 0:
...         return -x
...     else
...         return x
...
>>> abs
<function abs at 0x104cbb880>
>>> type(abs)
<class 'function'>
>>>
```

FUNCTION VARIABLE ASSIGNMENT

You can assign (and reassign) variables to functions

```
>>> f = square
>>> g = abs
>>> f
<function square at 0x104cbb7e0>
>>> g
<function abs at 0x104cbb880>
>>> f(-3)
9
>>> g(-3)
3
>>> g = f
>>> g
<function square at 0x104cbb7e0>
>>> g(-3)
9
>>>
```

FUNCTION VARIABLES IN A SCRIPT

This can be quite powerful. Here is a script that uses one:

```
def square(x): return x * x
def cube(x): return x ** 3

print("Which function would you like to apply?")
which = input("Enter 'square' or 'cube': ")
if which == "square":
    f = square
else:
    f = cube

x = int(input("Enter the function's input: "))

y = f(x)
print(which + "(" + str(x) + ") is " + str(y))
```

THE HIGHER-ORDER FUNCTION FEATURES OF PYTHON

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

Generally:

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

- Assign variables to be function objects,
- Pass functions/procedures as arguments to other functions/procedures.

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.***

A HIGHER-ORDER PROCEDURE

Let's invent a procedure that reports a function's value

```
def report_eval(name,f,x):
    ????
```

Here is how I'd like it to work:

```
>>> report_eval(“abs”,abs,-5)
The value of abs(-5) is 5.
>>> report_eval(“abs”,abs,3)
The value of abs(3) is 3.
>>> report_eval(“square”, square, -5)
The value of square(-5) is 25.
>>> report_eval(“square”, square, 3)
The value of square(3) is 9.
```

A HIGHER-ORDER PROCEDURE

This procedure reports a function's value:

```
def report_eval(name, f, x):

    # evaluate f at x
    y = f(x)

    # build the report string
    it = name + “(“ + str(x) + “)”
    that = str(y)
    s = “The value of “ + it + “ is “ + that + ”.”

    # output the report string
    print(s)
```

Here is it in use:

```
>>> report_eval(“abs”, abs, -5)
The value of abs(-5) is 5.
>>> report_eval(“square”, square, 3)
The value of square(3) is 9.
```

ANOTHER HIGHER-ORDER PROCEDURE

How about this procedure?

```
def natfun_report(name, natfun, n):
    ????
```

Here is how I'd like it to work:

```
>>> sequence_report("square", square, 9)
 n | square(n)
---+---------
 1 | 1
 2 | 4
 3 | 9
 4 | 16
 5 | 25
 6 | 36
 7 | 49
 8 | 64
 9 | 81
```

A SEQUENCE REPORTER

Here is the code for it:

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

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:

```
>>> def multiply(x,y):
...     return x * y
...
>>> abcde(multiply,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
```

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
```

HIGHER-ORDER FUNCTION FEATURES

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

Generally:

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

- Assign variables to be function objects, √
- Pass functions/procedures as arguments to other functions/procedures. √

HIGHER-ORDER FUNCTION FEATURES

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

Generally:

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

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

HIGHER-ORDER FUNCTION FEATURES

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

Generally:

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

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

We will talk about these features later.

PROJECT 1

AUTOMATING A DICE STRATEGY GAME

JIM FIX, REED COLLEGE CSCI 121