PROCEDURES AND NONE

LECTURE 03-2

JIM FIX, REED COLLEGE CSCI 121

MORE EXAMPLES: MIXING TYPES WITH WHAT'S RETURNED

- The function below determines whether an integer `rating` is from 1 to 10.
- It returns either the integer or a string:

```
def assessRating(rating):
    if (rating > 0) and (rating <= 10):
        return rating
    else:
        return “not a rating”
```

- Below is it in use:

```
>>> assessRating(3)
3
>>> assessRating(11)
'not a rating'
```

MISSING CASES?

- What happens if you (accidentally) forget a case?

```
def example(value):
    if value > 0:
        return "positive"
    elif value < 0:
        return "negative"
```

- What happens in the missing case?

```
>>> example(3)
'positive'
>>> example(-4)
'negative'
>>> example(0)
????
```

MISSING CASES?

- What happens if you (accidentally) forget a case?

```
def example(value):
    if value > 0:
        return "positive"
    elif value < 0:
        return "negative"
```

- What happens in the missing case?

```
>>> example(3)
'positive'
>>> example(-4)
'negative'
>>> example(0)
>>>
```

MISSING CASES

- What happens if you (accidentally) forget a case:

```
def example(value):
    if value > 0:
        return "positive"
    elif value < 0:
        return "negative"
```

- What happens in the missing case?

```
>>> print(example(3))
positive
>>> print(example(4))
negative
>>> print(example(0))
None
>>>
```

MISSING CASES

- What happens if you (accidentally) forget a case:

```
def example(value):
    if value > 0:
        return "positive"
    elif value < 0:
        return "negative"
```

- What happens in the missing case?

```
>>> print(example(3))
positive
>>> print(example(4))
negative
>>> print(example(0))
None
>>>
```

- There is a special Python value `None` that is implicitly returned.
- Confusingly, the interpreter does not display the `None` value.

MISSING CASES

- What happens if you (accidentally) forget a case:

```
def example(value):
    if value > 0:
        return "positive"
    elif value < 0:
        return "negative"
```

- What happens in the missing case?

```
>>> print(example(3))
positive
>>> print(example(4))
negative
>>> print(example(0))
None
```

- There is a special Python value `None` that is implicitly returned.
- *Make sure in your functions you've an explicit* `return` *for every case!*

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)
+----+
|    |
+----+
>>>
```

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
>>> greetThenReturnNone("Jim")
Hello, Jim.
>>> print(greetThenReturnNone("Jim"))
Hello, Jim.
None
```

FUNCTIONS VS. PROCEDURES

- 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).
- "**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.
  - When a procedure's work is done, Python continues executing after the line where it was called. (Control "jumps" then returns.)

SUMMARY OF USER-DEFINED PROCEDURES

- Procedures are like functions, defined using `def`.
  - They perform some work but don't return a value.
- A ~~function's~~ procedure's code consists of an indented ***body*** of statements.
  - These statements are ones like the top-level ones used in scripts.
- The ~~function's~~ procedure's lines of code use its ***parameter*** variables.
- The last line executed is a `return` statement with no value.
  - Or `return` may be missing and so it is an *implicit one.*
  - ***Be careful:*** a function might have the bug of an implicit `return`.
- Other earlier lines can also have a `return` statement.
  - They lead to an imeediate exit back to the calling code.

SUMMARY OF USER-DEFINED PROCEDURES

- Procedures are like functions, defined using `def`.
  - They perform some work but don't return a value.
- A ~~function's~~ procedure's code consists of an indented ***body*** of statements.
  - These statements are ones like the top-level ones used in scripts.
- The ~~function's~~ procedure's lines of code use its ***parameter*** variables.
- The last line executed is a `return` statement with no value.
  - Or `return` may be missing and so it is an *implicit one.*
  - ***Be careful:*** a function might have the bug of an implicit `return`.
- Other earlier lines can also have a `return` statement.
  - They lead to an imeediate exit back to the calling code.

- ***^^^ This is my terminology, not Python's.***
  - Sometimes procedures/functions do work ***and*** return a value.

FUNCTION VARIABLES

LECTURE 03-2
(QUICK) 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 mypoly(x):
    return x**4 – 8*x**3 + 6*x – 4

min_so_far = mypoly(3)
where_seen = 3
i = 4
while i <= 10:
    if mypoly(i) < min_so_far:
        min_so_far = mypoly(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 *function being checked*:

```
def mypoly(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 some_function(i) < min_so_far:
            min_so_far = some_function(i)
            where_seen = i
        i = i + 1
    return where_seen

print(argument_for_min(mypoly, 3, 10))
print(argument_for_min(mypoly, -20, 5))
print(argument_for_min(mypoly, 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 some_function(i) < min_so_far:
            min_so_far = some_function(i)
            where_seen = i
        i = i + 1
    return where_seen

def mypoly(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(mypoly, 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

How about this procedure?

```
def sequence_report(name, intfun, upto):
    ????
```

Here is how I'd like it to work:

```
>>> sequence_report(“square”, square, 9)
 i | 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 sequence_report(name, intfun, upto):
    print(" i | " + name + "(upto)")
    print("-"*3 + "+" + "-"*(len(name)+5))
    i = 1
    while i <= upto:
        print(" "+str(i)+" | “+str(intfun(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
```

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
```

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
```

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: ROLL100

AUTOMATING A DICE STRATEGY GAME

JIM FIX, REED COLLEGE CSCI 121