

---

# CS 111: Program Design I

## Lecture # 7: First Loop, Web Crawler, Functions

---

Robert H. Sloan & Richard Warner  
University of Illinois at Chicago  
September 17, 2019

What will the value of z be after this code runs?

```
def foo(x):  
    if x != 3:  
        return 1  
    return 2  
  
z = foo(-1)
```

- A. 1
- B. 2
- C. This will cause an error

What will the value of z be after this code runs?

```
def foo(x):  
    if x != 3:  
        print(1)  
    return 2  
  
z = foo(-1)
```

- A. 1
- B. 2
- C. This will cause an error or odd unexpected result

# shifting by k for *very* short strings

```
def shift(s, k):  
    if len(s) == 1:  
        return rotate(s, k)  
    if len(s) == 2:  
        ans = (rotate(s[0], k) +  
               rotate(s[1], k))  
        return ans  
    print("Sorry, can't help you")
```

# From length 2 to 3

```
def shift(s, k):  
    if len(s) == 1:  
        return rotate(s, k)  
    if len(s) == 2:  
        ans = (rotate(s[0], k) +  
              rotate(s[1], k))  
        return ans  
    if len(s) == 3:  
        return (rotate(s[0], k) +  
              rotate(s[1], k) +  
              rotate(s[2], k))  
    print("Sorry, can't help you")
```

# Or even 4

```
def shift(s, k):  
    if len(s) == 1:  
        return rotate(s, k)  
    if len(s) == 2:  
        return rotate(s[0], k) + rotate(s[1], k)  
    if len(s) == 3:  
        return (rotate(s[0], k) + rotate(s[1], k)  
                + rotate(s[2], k))  
    if len(s) == 4:  
        return (rotate(s[0], k) + rotate(s[1], k)  
                + rotate(s[2], k) + rotate(s[3], k))  
print("Sorry, can't help you")
```

---

# But...

- This will be a real drag even for 140 character tweets
- Imagine the 5-page report with 15,000 characters. . . .
- We need **for loops**
  - for loops allow us to do same thing for every item in a sequence!



# **FIRST LOOK AT FOR LOOPS**



---

# for loop basic idea

```
for c in st:
```

```
    <body that refers to c>
```

- execute body `len(st)` times, once each with `c` being each character of string `st` in order
- (In general, `st` could be any sequence, e.g., a list and `c` becomes each item of `st` once)

# Example: d detector

```
In [1]: d('I would like dodecarchy in the US')
```

```
Out[1]: 3
```

```
In [2]: d('I am against the letter after c')
```

```
Out[2]: 0
```

```
def d(input):  
    counter = 0  
    for symbol in input:
```

# Example: d detector

```
In [1]: d('I would like dodecarchy in the US')
```

```
Out[1]: 3
```

```
In [2]: d('I am against the letter after c')
```

```
Out[2]: 0
```

```
def d(input):  
    counter = 0  
    for symbol in input:  
        if symbol == 'd':  
            counter = counter + 1  
    return counter
```

We choose the name of a variable...

... and we provide a sequence

# This will print?

```
for x in "0123":  
    print(x)
```

A. 0

B. 0

3

C. 0

1

2

3

D. 0

1

2

3

4

E. This will  
run  
forever

# Lab Hint 1: Build up answer in for loop

- Very often *build up answer* to return *inside* for loop *and return it outside* loop, after its end:

```
def d(input):  
    counter = 0  
    for symbol in input:  
        if symbol == 'd':  
            counter = counter + 1  
    return counter
```

# Example

- Return string with lower-case characters replaced by X, all other left unchanged

```
def x_it(input):  
    """Example for lecture slides"""  
    answer = '' #Empty string  
    for c in input:  
        next = c  
        if next.islower():  
            next = 'X'  
        answer = answer + next  
    return answer
```

# Using else instead

```
def x_else_it(input):  
    """Example with if-else instead of  
    if"""  
    answer = '' #Empty string  
    for c in input:  
        if c.islower():  
            answer = answer + 'X'  
        else:  
            answer = answer + c  
    return answer
```

# Lab Hint(s) #2/Reminder

```
word = 'hi'
```

```
word.upper() → "HI"
```

```
"h".isalpha() → True
```

```
'h'.islower() → True
```

```
'h'.isupper() → False
```



---

Winter is coming

# ~~Winter is~~ Midterm & Project coming

- Midterm 1 next week Tuesday
  - Will cover material from assigned reading so far, further details given in lecture, and legal material on lecture
  - Some of Thursday: Review
- Lab out tomorrow (due Friday), which will be part of
- Project 1: Full Caesar and Vigenère ciphers
  - Not due until 1 week from Sunday (but start)

---

Towards Crawling the Web

# **(MORE ABOUT) FUNCTIONS**

# Web crawler

- One long-term goal of course: build and understand **web crawler**, program that will visit every page reachable from given start web page
  - Key component of, e.g., search engine
- Many pieces, somewhat complicated
  - Need an organizing principle: functions!
  - Also need to do things over and over: iteration
- Will return to crawler from time to time

# Functions: definition & use

- Can do 2 things with function: *Define* it; *Call* it

## Definition:

```
def fn_name(parameters):
```

- E.g., `def string_multiply(my_string, num):`

## Call (use)

```
fn_name(parameters)
```

- E.g, `string_multiply('hi', 3)`

- Runs function `fn_name` on parameters

---

Note: Definition *must* have some indented code after the `def fn_name():` line

- This is *not* a legal function definition:

```
def nothingness():
```

- But

```
def nothingness():  
    return
```

- is legal (though useless) function definition

# Input parameters (1)

- Most functions have  $\geq 1$  input parameters (though legal & sometimes appropriate to create function with no input parameters).
- Example of built-in function (technically specifically a *method* function) with zero inputs: String method `upper()`:

```
word = "hi"
```

```
word.upper() → "HI"
```

# Return values

- function may or may not **return** a value
- If need terminology, call function that returns value **fruitful** function; function that doesn't non-fruitful function
- If (and only if) function returns value, legal to assign name to (return value of) function call:

```
x = fruitful_fn_name(inputs)
```



# Example

- Say we want to find absolute value of a number (say -3)
- There is built-in Python function called `abs` that finds absolute value

```
In [1]: x = abs(-3)
```

```
In [2]: x
```

```
Out[2]: 3
```

---

# Most famous built-in non-fruitful function

- `print()`
- `print` doesn't return any value.
  - We don't use `print` (or any non-fruitful function) for its return value but for some other reason

---

functions you write that return something

- **Must** *include a line that begins*

return

---

# function flow

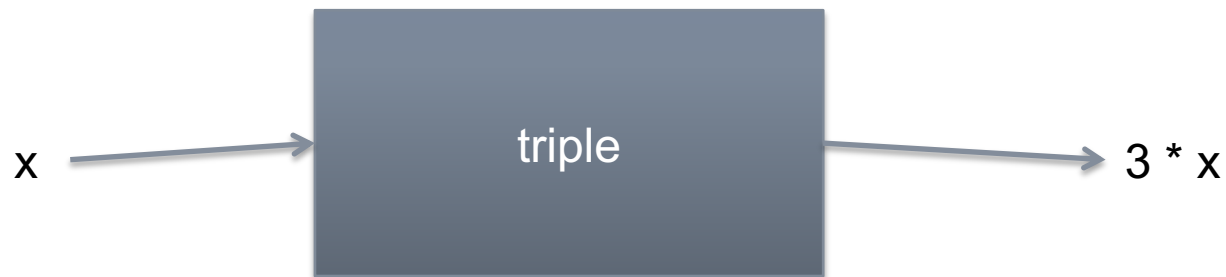
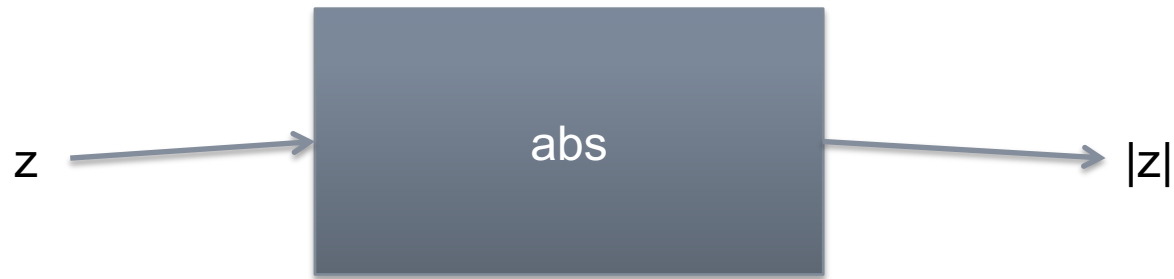
- A function's execution ends either when
  1. A return statement is executed, or
  2. Last line of code is executed
- whichever comes first

# What is wrong with this function?

```
def triple(x):  
    return 3 * x  
    print("Triple the input is", 3*x)
```

- A. You should never use a print statement in a function
- B. You must calculate the value of  $3*x$  before you return it
- C. You should not have statements after the return
- D. A function can return string types but not a number
- E. Nothing

# Function as input-output box



# Parameters (1)

(actual) parameter

>>> y = abs(-3)

>>> y

3

# Parameters (2)


- Parameters in the ()s in def statement called *formal parameters*
- Formal parameters are (like) variables. they're the thing that changes inside function
- Functions have 0 or more parameters
- Value(s) in function call: *actual parameter(s)*
  - Must be same number as formal parameters
  - Could be variables and/or literal values



# Formal vs. actual parameters

```
def triple(x):  
    return 3 * x
```

formal parameter



```
In [1]: n = 17
```

```
In [2]: triple(n)
```

```
Out[2]: 51
```

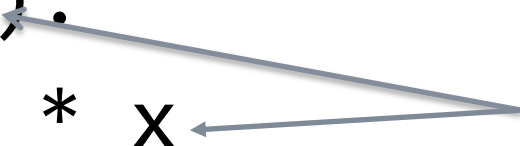
```
In [3]: 4 + triple(20)
```

```
Out[3]: 64
```

# Formal vs. actual parameters

```
def triple(x):  
    return 3 * x
```

formal parameter  
(which happens to  
be x in this example)



```
In [1]: n = 17
```

```
In [2]: triple(n)
```

```
Out[2]: 51
```

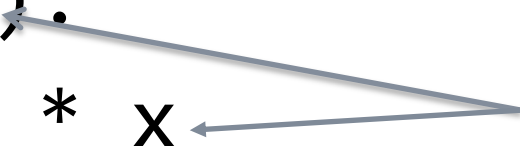
```
In [3]: 4 + triple(20)
```

```
Out[3]: 64
```

# Formal vs. actual parameters

```
def triple(x):  
    return 3 * x
```

formal parameter  
(which happens to  
be x in this example)



```
In [1]: n = 17
```

```
In [2]: triple(n)
```

```
Out[2]: 51
```

```
In [3]: 4 + triple(20)
```

```
Out[3]: 64
```

actual parameter



# Parameters and function execution

- Like functions in high-school Algebra 2:
  - At time function called, formal parameter takes on value of actual parameter
- Algebra 2:
  - if  $f(x) = 3x$ , then  $4 + f(20) = 64$ 
    - and implicitly at least, the formal parameter  $x$  took on the value 20.
- Python:  
`4 + triple(3)`
  - formal  $x$  in def of `triple()` bound to 3 for length of run of `triple()`

# In even more detail

```
def triple(x):  
    return 3 * x
```


← formal parameter

```
>>> 4 + triple(20)
```

# In even more detail

```
def triple(x):  
    return 3 * x
```

formal parameter  
x bound to 20



```
>>> 4 + triple(20)
```

- At point where `triple(20)` is called, value 20 is assigned to `triple`'s internal `x` parameter, multiplication is done getting value 60, number 60 is returned (and `triple` is done), and interpreter (command line) adds 4 and 60