# CS 111: Program Design I

## Lecture 8: Functions (cont.), if, Midterm 1

Robert H. Sloan & Richard Warner

University of Illinois at Chicago

September 19, 2019

# This Class

- More on functions, if else
- Any fast Lab 4 questions
- Discuss midterm topics
- Go over practice examples

- Answer any questions

# FUNCTIONS CONCLUDED (AT LEAST FOR NOW)

# 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

# What will print after this code?

```
def add_three(x):
    x = x + 3
    return x

>>> x = 5
>>> n = add_three(x)
>>> print(x)
```

A. 3

B. 5

C. 8

D. This will cause an error

# What will print after this code?

```
def add_three(x0):
    x0 = x0 + 3
    return x0

>>> x = 5
>>> n = add_three(x)
>>> print(x)
```

A. 3

B. 5

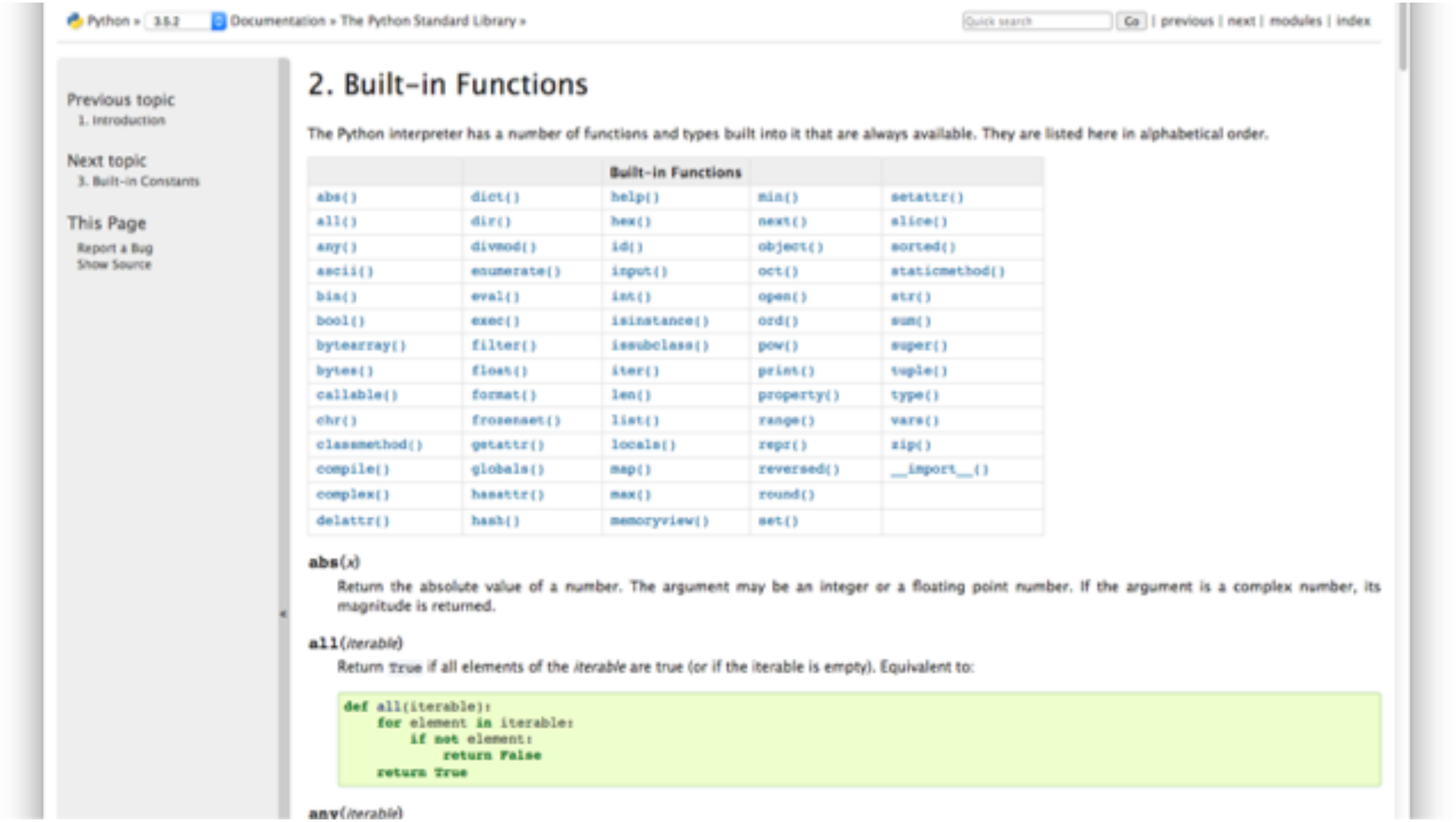
C. 8

D. This will cause an error

# Parameter of function is local to function

- Ultimately because of how things are stored in memory
- Name x in parameter is distinct from any x after end of function running
- More generally, formal parameters and and variables assigned to inside a function are *local variables* to that function, and they exist only when function is running

# How do I know which functions exist? Python documentation



## 2. Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

| | | Built-in Functions | | |
|---|---|---|---|---|
| abs() | dict() | help() | min() | setattr() |
| all() | dir() | hex() | next() | slice() |
| any() | divmod() | id() | object() | sorted() |
| ascii() | enumerate() | input() | oct() | staticmethod() |
| bin() | eval() | int() | open() | str() |
| bool() | exec() | isinstance() | ord() | sum() |
| bytearray() | filter() | issubclass() | pow() | super() |
| bytes() | float() | iter() | print() | tuple() |
| callable() | format() | len() | property() | type() |
| chr() | frozenset() | list() | range() | vars() |
| classmethod() | getattr() | locals() | repr() | zip() |
| compile() | globals() | map() | reversed() | __import__() |
| complex() | hasattr() | max() | round() | |
| delattr() | hash() | memoryview() | set() | |

**abs**(*x*)

Return the absolute value of a number. The argument may be an integer or a floating point number. If the argument is a complex number, its magnitude is returned.

**all**(*iterable*)

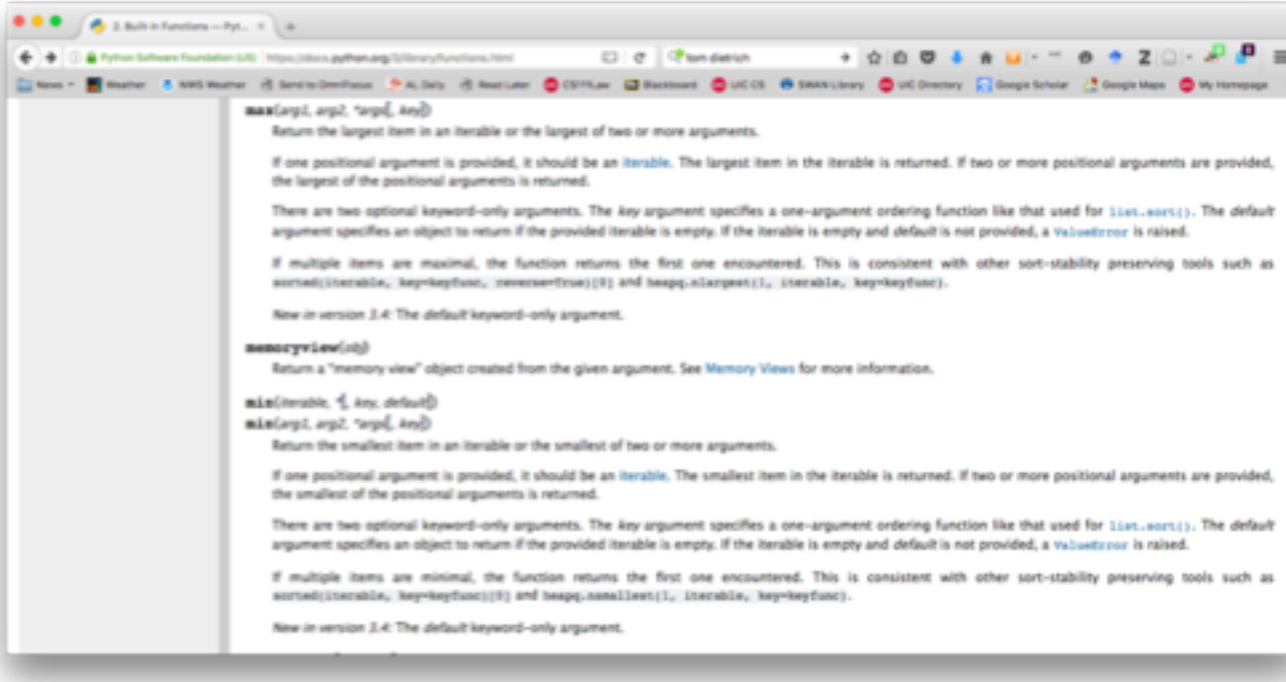
Return `True` if all elements of the *iterable* are true (or if the iterable is empty). Equivalent to:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

**any**(*iterable*)

# What should I do to make z the smaller of integers x and y?

```
x = 10
y = 3
```



**max**(*arg1, arg2, *args*[, *key*])

Return the largest item in an iterable or the largest of two or more arguments.

If one positional argument is provided, it should be an iterable. The largest item in the iterable is returned. If two or more positional arguments are provided, the largest of the positional arguments is returned.

There are two optional keyword-only arguments. The *key* argument specifies a one-argument ordering function like that used for `list.sort()`. The *default* argument specifies an object to return if the provided iterable is empty. If the iterable is empty and *default* is not provided, a `ValueError` is raised.

If multiple items are maximal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as `sorted(iterable, key=keyfunc, reverse=True)[0]` and `heapq.nlargest(1, iterable, key=keyfunc)`.

*New in version 3.4:* The *default* keyword-only argument.

**memoryview**(*obj*)

Return a "memory view" object created from the given argument. See Memory Views for more information.

**min**(*iterable*, *[, *key, default*])
**min**(*arg1, arg2, *args*[, *key*])

Return the smallest item in an iterable or the smallest of two or more arguments.

If one positional argument is provided, it should be an iterable. The smallest item in the iterable is returned. If two or more positional arguments are provided, the smallest of the positional arguments is returned.

There are two optional keyword-only arguments. The *key* argument specifies a one-argument ordering function like that used for `list.sort()`. The *default* argument specifies an object to return if the provided iterable is empty. If the iterable is empty and *default* is not provided, a `ValueError` is raised.

If multiple items are minimal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as `sorted(iterable, key=keyfunc)[0]` and `heapq.nsmallest(1, iterable, key=keyfunc)`.

*New in version 3.4:* The *default* keyword-only argument.

A. z = max(x,y)
B. z = min(x,y)
C. z = memoryview(x)
D. z = memoryview(y)

# Coming attraction: Additional built-in functions from modules

- Useful for certain kinds of things, e.g., math, internet, making graphs, available in *modules* that must be imported before can be used
- Will discuss some later as needed

# functions for strings

- Strings examples of "built-in class" and String class comes with some built-in functions (and class functions also called methods). (A little more much later)
- Same as other built-in functions *except calling syntax* is .fn_name
  - `st = "60 votes for cloture"`
  - `st.upper()` → `"60 VOTES FOR CLOTURE"`
  - `st.find("V")` → `-1`
  - `st.find("v")` → `3`

# Some Notes on Programming Style

- Remember: Code needs to be understood by both computers and *people*

- Should try to make code as easy to read as possible

- Pro tip: This will make it easier for our TAs and I to give you partial credit on assignments, exams, etc

# Good Python Programming Style

- Meaningful variable & function names
  - Generally starting with lower-case letter
  - Python style: Use underscore not camel case for 2-word names: two_word (*not* twoWord)
- Blank line between functions
- Use of docstring to briefly describe input-output behavior of function
- And, of course, be very careful with indentation

# Why functions instead of e.g., cut & paste same code

- Code length (repeating same thing)
- Bugs: If there's bug or error, replicated in multiple places
- If we want to change something, need to change it in every copy

# Analogy

- Imagine you are writing cake cookbook with 17 recipes that use buttercream frosting

- Do you put the buttercream instructions in each of the 17 recipes?

# Even if a function used only once

- Helps modularize code and make it easier for humans to read and understand

# functions as aid to problem solving

- Problem solving strategy:
  - Describe how to solve your problem assuming whenever you like that you have a function to do some of the work
    - Figure out what input-output behavior it needs
    - Then write those functions
    - This is called functional decomposition
    - Tend to use it on slightly larger problems than we have worked on so far; will revisit

# Testing. . . .

- Early and often
- Each function
- The Called before the callers
- Edge cases

# Recall: Sequential coding elements

- Generally Python statements run one at a time, in order we write them in

- Assignment statements
- Function calls

- *But* if, for, (and while) change execution order

# if *else* conditionals

```
if <condition>:
    <body1>
else:
    <body2>
```

- If the condition is True, then run <body1>; otherwise, run <body2>

# What will this print

```
x = "Roberts"
if len(x) < 3:
    print("Hi!")
else:
    print("Bye!")
```

A. Nothing

B. "Hi!"

C. "Bye!"

D. "Hi!" and "Bye!"

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

```
def foo(x):
    if x < 3:
        return 1
    else:
        return 2

z=foo(-1)
```

A. 1

B. 2

C. 3

D. -1

E. This will cause an error

# Will function foo ever return 3?

```
def foo(x):
    if x < 3:
        return 1
    else:
        return 2
    return 3
```

A. Yes B. No

# Will function foo ever return 3?

```
def foo(x):
    if x < 3:
        return 1
    else:
        return 2
        return 3
```

A. Yes B. No

# Will function foo ever return 3?

```
def foo(x):
    if x < 3:
        return 1
    else:
        print(2)
        return 3
```

A. Yes B. No

# MIDTERM REVIEW

# Recall: Learning programming…

1) *Expect it to be different!*

2) *Don't feel you need to memorize it*

3) *Immersion == Experimentation*

# The Secret of Happiness is…

(in programming)

- Don’t memorize!
- Look at examples of similar problems
- Experiment
- Syntax that looks weird now will become second nature soon

# Bring UIC ID to Midterm Tuesday!

# Midterm I: Topics Covered

- Objects & Variables
- Mathematical operators
- Statements
- Types
- Strings
  - []s (i.e., indexing), slicing, .find()
- Functions
- Relational, Boolean, and membership operators
- if-else
- simple for over a string's characters

# Midterm I: Topics, continued

- What is an algorithm, computer, RAM, etc.
- Encryption
- Encryption keys and government access

# Any general questions?

# What type of variable would you use to store the length of a plaintext?

A. int

B. float

C. list

D. boolean

E. string

# What type of variable would you use to store the length of a plaintext?

[✓ A. int]

B. float

C. list

D. boolean

E. string

What type would you use for a variable to store the fraction of Chicago wards with more than 20 homicides per 100,000 population?

A. int

B. float

C. list

D. boolean

E. string

What type would you use for a variable to store the fraction of Chicago wards with more than 20 homicides per 100,000 population?

A. int

[✓ B. float]

C. list

D. boolean

E. string

# What type would you use for a variable to store whether a plaintext contains any space characters?

A. int

B. float

C. list

D. boolean

E. string

What type would you use for a variable to store whether a plaintext contains any space characters?

A. int

B. float

C. list

D. [✓ boolean]

E. string

# What type would you use for a variable for that plaintext?

A. int

B. float

C. list

D. boolean

E. string

## What type would you use for a variable for that plaintext?

A. int

B. float

C. list

D. boolean

E. [✓ string]

# What *type* is expression on the last line?

A. int

[✓ B. float]

C. list

D. boolean

E. string

```
x = 13
z = 2.5
z * x
```

## Suppose you have the following function defined:

```
def square(x):
    return x**2
```

Write a function that takes integers x and y and prints the larger of $x^2$ and $y^2$. Don't forget the docstring!

# key ="LEMON"

- Write an expression that returns the first character in key
- Write an expression that returns the last character in key
- Write an expression that returns every other position in this key, starting with the first
- Write an expression that returns key reversed

# key ="LEMON"

- Write an expression that returns the first character in key key[0]
- Write an expression that returns the last character in key key[-1] *or* key[len(key) - 1]
- Write an expression that returns every other position in this key, starting with the first key[::2]
- Write an expression that returns key reversed key[::-1]

# The 4th and 5th Amendment

(a)The 4th and 5th Amendment protect against government searches.

(b)The 4th Amendment protects against self-incrimination.

(c)The 5th Amendment protects against government searches.

[✓ (d)The 4th Amendment protects against government searches and the 5th Amendment protects against self-incrimination.]