Lecture 3

Strings, Functions, & Modules

String: Text as a Value

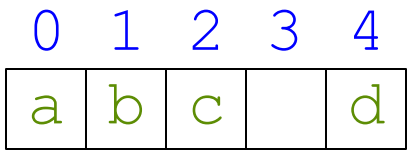
- String are quoted characters
  - `'abc d'` (Python prefers)
  - `"abc d"` (most languages)
- How to write quotes in quotes?
  - Delineate with "other quote"
  - **Example**: `" ' "` or `' " '`
  - What if need both `"` and `'` ?
- **Solution**: escape characters
  - Format: \ + letter
  - Special or invisible chars

Type: `str`

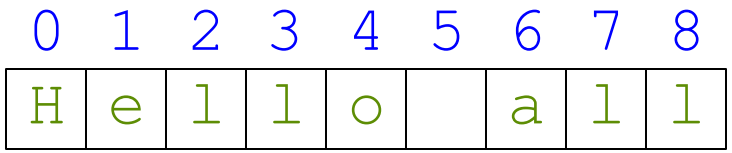
Char	Meaning
`\'`	single quote
`\"`	double quote
`\n`	new line
`\t`	tab
`\\`	backslash

String are Indexed

- s = 'abc d'



- s = 'Hello all'



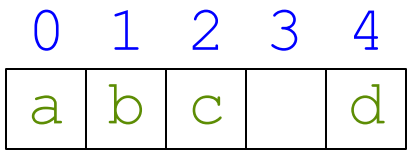
- Access characters with []
  - s[0] is 'a'
  - s[4] is 'd'
  - s[5] causes an error
  - s[0:2] is 'ab' (excludes c)
  - s[2:] is 'c d'
- Called "string slicing"

- What is s[3:6]?

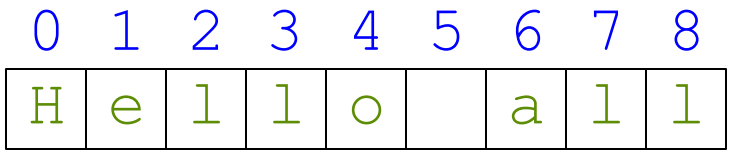
A: 'lo a'
B: 'lo'
C: 'lo '
D: 'o '
E: I do not know

String are Indexed

- s = 'abc d'



- s = 'Hello all'



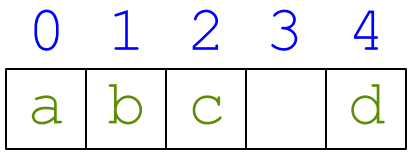
- Access characters with []
  - s[0] is 'a'
  - s[4] is 'd'
  - s[5] causes an error
  - s[0:2] is 'ab' (excludes c)
  - s[2:] is 'c d'
- Called "string slicing"

- What is s[3:6]?

A: 'lo a'
B: 'lo'
C: 'lo ' **CORRECT**
D: 'o '
E: I do not know

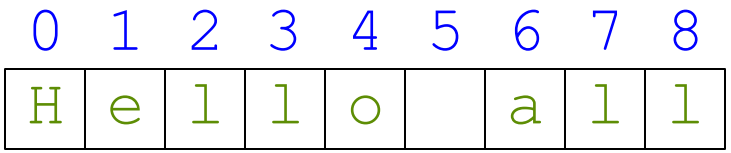
String are Indexed

- s = 'abc d'



- Access characters with []
  - s[0] is 'a'
  - s[4] is 'd'
  - s[5] causes an error
  - s[0:2] is 'ab' (excludes c)
  - s[2:] is 'c d'
- Called "string slicing"

- s = 'Hello all'

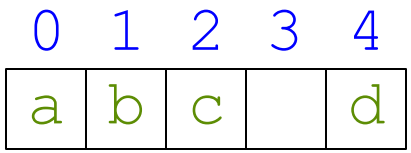


- What is s[:4]?

A: 'o all'
B: 'Hello'
C: 'Hell'
D: Error!
E: I do not know

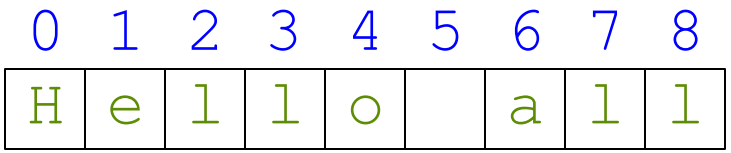
String are Indexed

- s = 'abc d'



- Access characters with []
  - s[0] is 'a'
  - s[4] is 'd'
  - s[5] causes an error
  - s[0:2] is 'ab' (excludes c)
  - s[2:] is 'c d'
- Called "string slicing"

- s = 'Hello all'



- What is s[:4]?

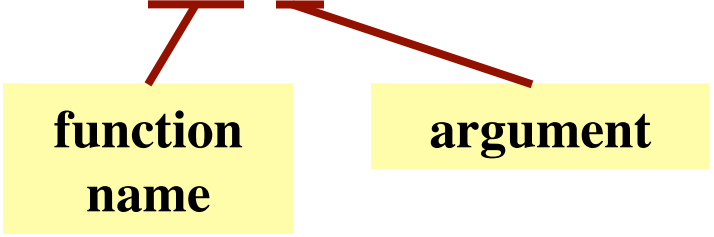
A: 'o all'
B: 'Hello'
C: 'Hell' **CORRECT**
D: Error!
E: I do not know

Other Things We Can Do With Strings

- **Operation** in: s_1 in s_2
  - Tests if s_1 "a part of" s_2
  - Say s_1 a *substring* of s_2
  - Evaluates to a bool
- **Examples**:
  - s = 'abracadabra'
  - 'a' in s == True
  - 'cad' in s == True
  - 'foo' in s == False
- **Function** len: len(s)
  - Value is # of chars in s
  - Evaluates to an int
- **Examples**:
  - s = 'abracadabra'
  - len(s) == 11
  - len(s[1:5]) == 4
  - s[1:len(s)-1] == 'bracadabr'

Function Calls

- Python supports expressions with math-like functions
  - A function in an expression is a **function call**
  - Will explain the meaning of this later
- Function expressions have the form **fun**(x,y,…)



- **Examples** (math functions that work in Python):
  - `round(2.34)`
  - `max(a+3,24)`

Arguments can be any **expression**

Built-In Functions

- You have seen many functions already
  - Type casting functions: `int()`, `float()`, `bool()`
  - Dynamically type an expression: `type()`
  - Help function: `help()`
- Getting user input: `raw_input()`
- `print <string>` is **not** a function call
  - It is simply a statement (like assignment)
  - But it is in Python 3.x: `print(<string>)`

Arguments go in (), but `name()` refers to function in general

Method: A Special Type of Function

- Methods are unique (right now) to strings
- Like a function call with a "string in front"
  - Usage: ***string*.*method***(*x*,*y*…)
  - The string is an *implicit argument*
- Example: upper()
  - `s = 'Hello World'`
  - `s.upper() == 'HELLO WORLD'`
  - `s[1:5].upper() == 'ELLO'`
  - `'abc'.upper() == 'ABC'`

Will see why we do it this way later in course

Examples of String Methods

- s_1.index(s_2)
  - Position of the first instance of s_2 in s_1
- s_1.count(s_2)
  - Number of times s_2 appears inside of s_1
- s.strip()
  - A copy of s with white-space removed at ends

- s = 'abracadabra'
- s.index('a') == 0
- s.index('rac') == 2
- s.count('a') == 5
- ' a b '.strip() == 'a b'

See Python Docs for more

Built-in Functions vs Modules

- The number of built-in functions is small
  - http://docs.python.org/2/library/functions.html
- Missing a lot of functions you would expect
  - **Example**: `cos()`, `sqrt()`
- **Module**: file that contains Python code
  - A way for Python to provide optional functions
  - To access a module, the `import` command
  - Access the functions using module as a *prefix*

Example: Module math

```
>>> import math
>>> math.cos(0)
1.0
>>> cos(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'cos' is not defined
>>> math.pi
3.141592653589793
>>> math.cos(math.pi)
-1.0
```

To access math functions

Functions require math prefix!

Module has variables too!

Example: Module math

```
>>> import math
>>> math.cos(0)
1.0
>>> cos(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'cos' is not defined
>>> math.pi
3.141592653589793
>>> math.cos(math.pi)
-1.0
```

To access math functions

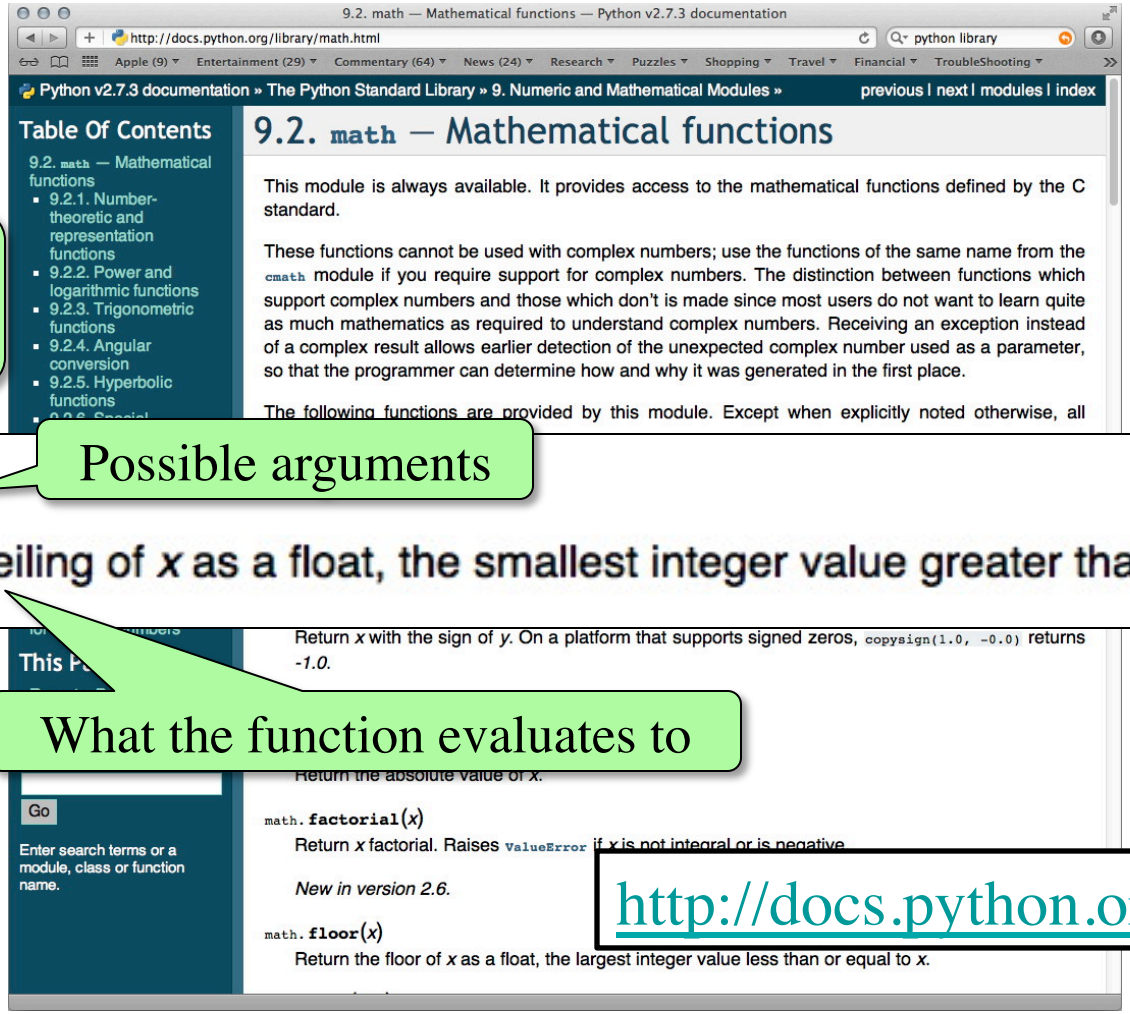
Functions require math prefix!

Module has variables too!

Other Modules

- io
  - Read/write from files
- random
  - Generate random numbers
  - Can pick any distribution
- string
  - Useful string functions
- sys
  - Information about your OS

Reading the Python Documentation



Using the from Keyword

```
>>> import math
>>> math.pi
3.141592653589793
>>> from math import pi
>>> pi
3.141592653589793
>>> from math import *
>>> cos(pi)
-1.0
```

Must prefix with module name

No prefix needed for variable pi

No prefix needed for anything in math

- Be careful using from!
- Namespaces are *safer*
  - Modules might conflict (functions w/ same name)
  - What if import both?
- **Example**: Turtles
  - Use in Assignment 4
  - 2 modules: turtle, tkturtle
  - Both have func. Turtle()

A String Puzzle (Extraction Practice)

- **Given**: a string with a parenthesis pair inside

```
s = 'labs are (usually) every week'
```

- **Goal**: expression for substring inside parentheses
  - **Step 1**: Find the open parenthesis

```
start = s.index('(')
```

  - **Step 2**: Store part of string **after** parenthesis in **tail**

```
tail = s[:start+1]
```

  - **Step 3**: Get the part of the tail **before** close parenthesis

```
tail[:tail.index(')')]
```

- **Given**: A string that is a list of words separated by commas, and spaces in between each comma:

```
pets = 'cat, dog, mouse, lion'
```

- **Goal**: Want second element with no spaces or commas. Put result inside of variable `answer`

Where, in the following sequence of commands, is there a (conceptual) error that prevents our goal?

A:
```
startcomma = info.index(',')
```

B:
```
tail = info[startcomma+1:]
```

C:
```
endcomma = tail.index(',')
```

D:
```
df = tail[:endcomma]
```

E: this sequence achieves the goal

- **Given**: A string that is a list of words separated by commas, and spaces in between each comma:

```
pets = 'cat, dog, mouse, lion'
```

- **Goal**: Want second element with no spaces or commas. Put result inside of variable `answer`

Where, in the following sequence of commands, is there a (conceptual) error that prevents our goal?

A: `startcomma = info.index(',')`

B: `tail = info[startcomma+1:]` +2 instead, or use

C: `endcomma = tail.index(',')`

D: `df = tail[:endcomma]` `tail[:endcomma].strip()`

E: this sequence achieves the goal