Intro to Computer Science

Previous

- Types
  - Introspection
  - Casting
- Variables
- Operators
- Input
- String methods

Readings	
Gaddis	• Chapter 2 • Chapter 8.3*
Python Std. Lib.	• Section 4.7.1

* "Searching, Manipulating"

Next

- Python plumbing
  - The `print` statement
  - Comments
  - Code style
- Sequences
  - Strings
  - Lists

Readings	
Gaddis	• Chapter 7.1—7.3, 7.5 • Chapter 8

Two ways to `print`

Concatenation

- Send a single argument to print
  - Generally a string
- String generated using the concatenation operator
- Requires an explicit cast of each variable

```
x = 10
print('Hey! ' + str(x) + ' men')
```

Must cast to avoid a TypeError

Arguments

- Send multiple arguments to print
  - Can be of various types
- Use commas to separate arguments
- `print` takes care of making a string

```
x = 10
print('Hey!', x, 'men')
```

Casting and concatenation happens implicitly

Using `print`'s Other Features

- There are several "options" that can be passed to print
  - How to separate *arguments* (previous slide)
  - How to end the line

Our string, presented as individual arguments

How we want to separate those arguments. Uses space by default

```
print('This', 'is', 'my', 'favourite', 'class', sep=';', end='!')
```

How we want to end the line. Uses newline by default

Comments (hashtag awesome)

- Comments are a part of good coding
- Serve two purposes
  1. Documentation (formal and notational)
  2. Code exclusion
- Use comments!
- Use # to create comments in Python
  - Once the parser sees #, it moves on to the next line

```
# Jerome White
# jsw7
# NYUAD CS101 FL 2014
x = ‘This little light of mine’
# x = “I’m gonna let it shine”
x[:4].lower().replace('s', 'p')[:1:-1] # 3.14 :)
```

Recall Strings

Values

- Any sequence of characters
- String literals come with quotes
  - “Hello World!”
  - ‘This is awesome’

Operations

Concatenation	+
Repetition	*

TypeError if these are violated!

- Concatenation operates on two strings
- Repetition operates on a string and an integer

String methods

- Functions available to string data types
- Returns a new string
  - Does not alter the string being called!

```
a = ‘Hello World’

x = a.lower()
y = a.count(‘l’)
z = a.replace(‘Hello’, ‘Goodbye’)
```

Strings are *Sequences* of Characters

- They look to us like normal words
- They look to Python like distinct elements

To us: x = ‘Hello’

To Python: x = | H | e | l | l | o |
|---|---|---|---|---|

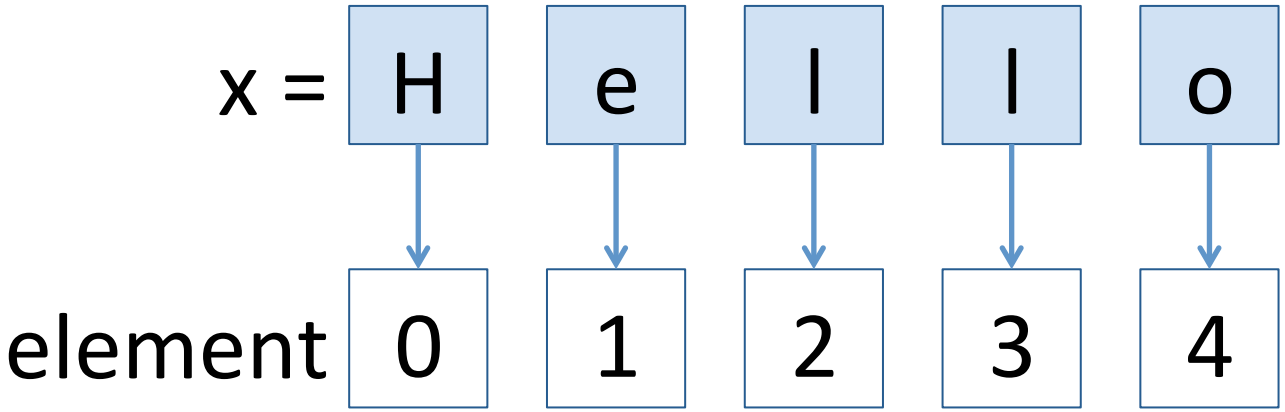
Subscript Notation

- Distinct elements means we can access individual characters
- Select the first index from the string:

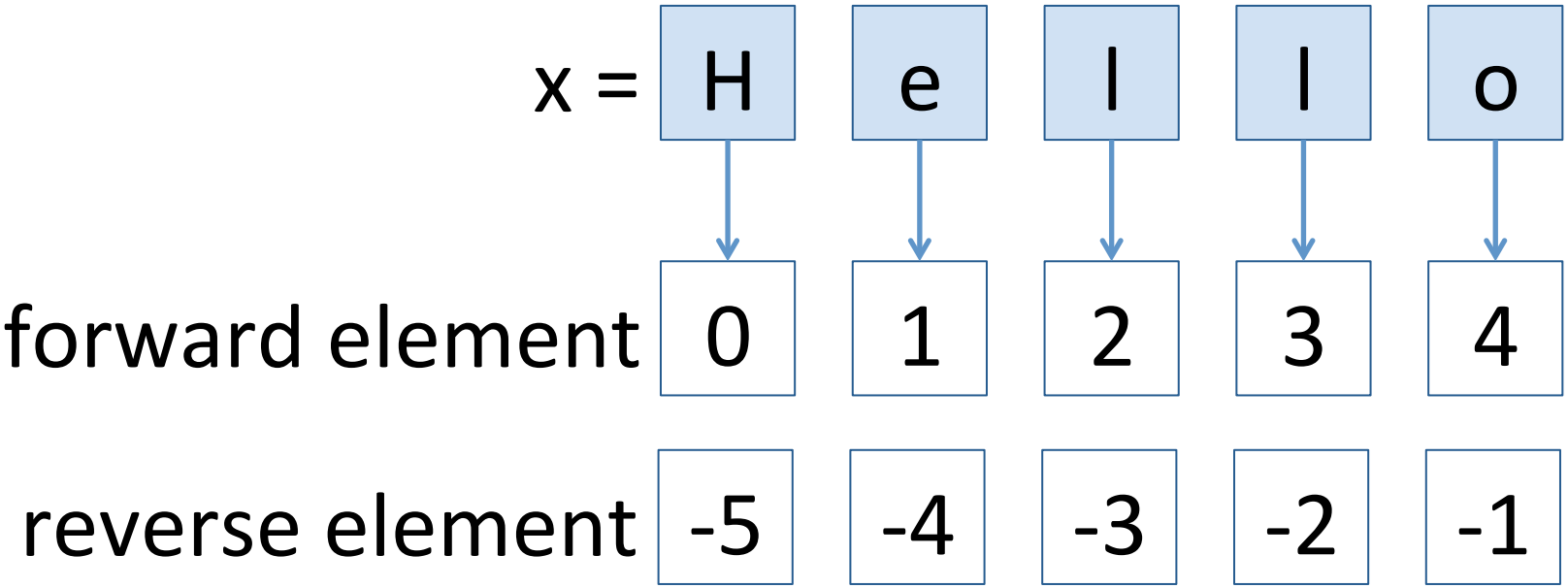
```
x = 'Hello'
print(x[1])
```

  – What's going on here?

Indexing Starts at 0!



Negative indexing



x[1] and x[-4] refer to the same element!

Bracketology

- Value inside the subscript must be an integer
- Negative values start from the end
  - Positive values: left-to-right
  - Negative values: right-to-left
- Values must be within range
  - Positive indices must be smaller than the string length
  - Negative values must be larger than the string length
  - IndexError otherwise

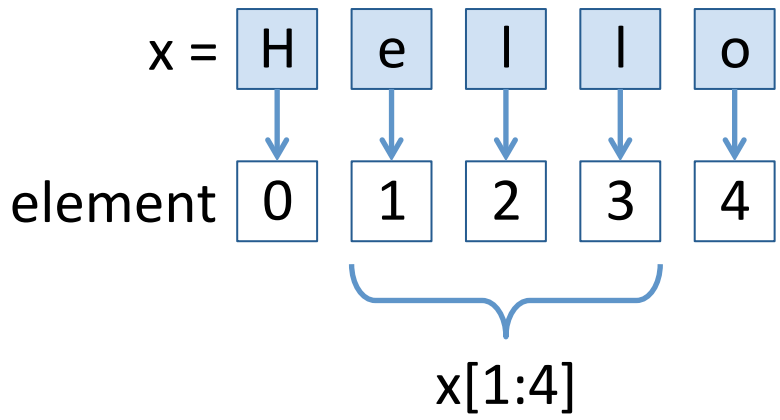
Stay in bounds!

- Some languages do not check the range of the bounds
  - Potentially better performance
  - But can be be extremely dangerous
- Because of the way memory works, buffer overruns can allow hackers to
  - Change program values
  - Change program execution
- Been a known problems since the 70's
- *Forms the basis of most computer exploits today*
- Most modern languages include inherent checks
  - Performance isn't worth it

Slicing

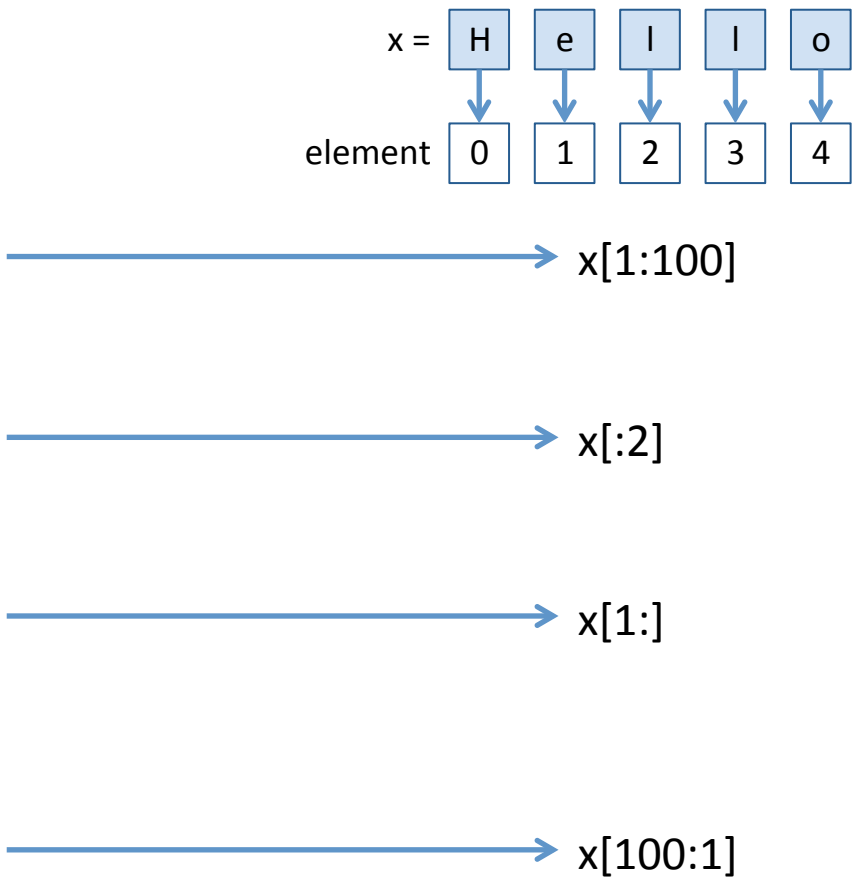
- A *slice* is a subset of the sequence
- Specified using a second value within the subscript
  - Colon notation
    [start:end]

    inclusive exclusive



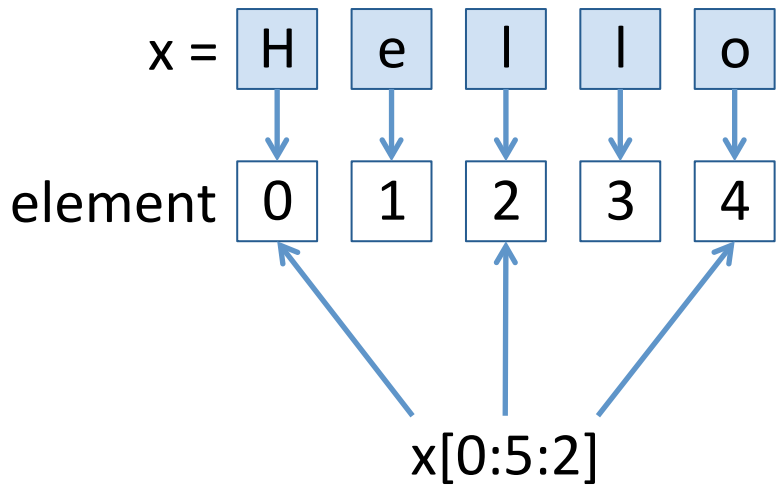
Notes about slicing

- The range does not have a bounds condition
  - Values larger than the length default to the length
- Omitting the first value defaults to 0
- Omitting the second defaults to the end of the string
- Out of order values result in an empty string



Stepping

- A third value specifies how to iterate over the range
  - More colon notation
- Slicing rules still apply
  - Can specify any combination of values
- A negative step reverses the string



Negative stepping

- A negative step value can be confusing
- Key is to think of negation as an output modifier
  - Indices and slices remain the same

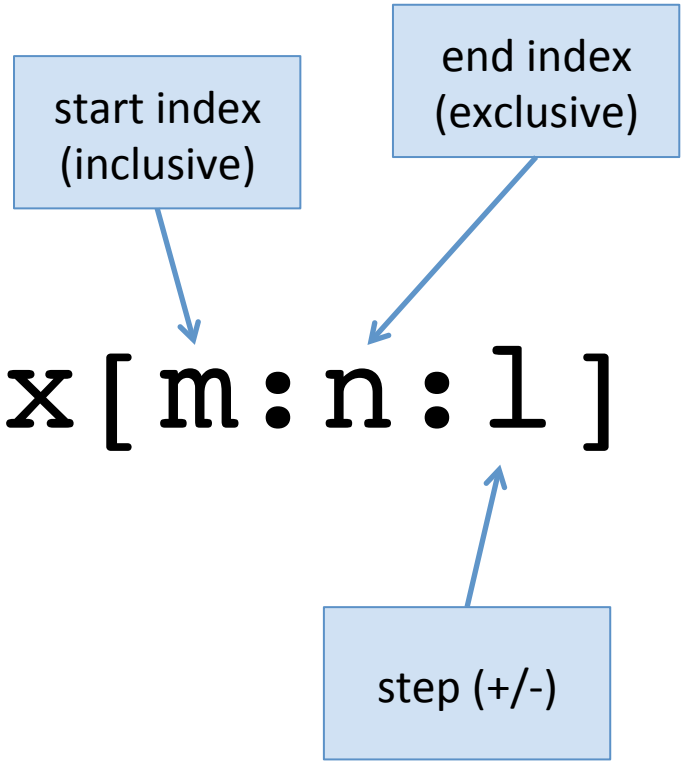
```
x = ‘Hello’
>>> x='hello'
>>> x[-2:-len(x)]
''
>>> x[-2:-len(x):-1]
'lle'
>>> x[3:-len(x):-1]
'lle'
>>> x[3:-len(x)]
’’
```

Discuss…

- Assign a string to a variable

  `x = 'mary had a little lamb'`

- Play around with various combinations of indexing, slicing, and stepping
  - Produce an error!
  - Build your own understanding!



An interesting example

```
x = '0123456789'
print(x[8:2:-2])
```

1. What does it produce?
2. Come up with three other combinations that produce the same thing:

$$x[a:b:c] \text{ where } \{a,b,c\} \neq \{8,2,-2\}$$

Lists

Values

- A sequence of *data*
- Use subscript to define
  - x = []
  - x = [1, 2, 3]
  - x = [1, ‘a’, 12.34]

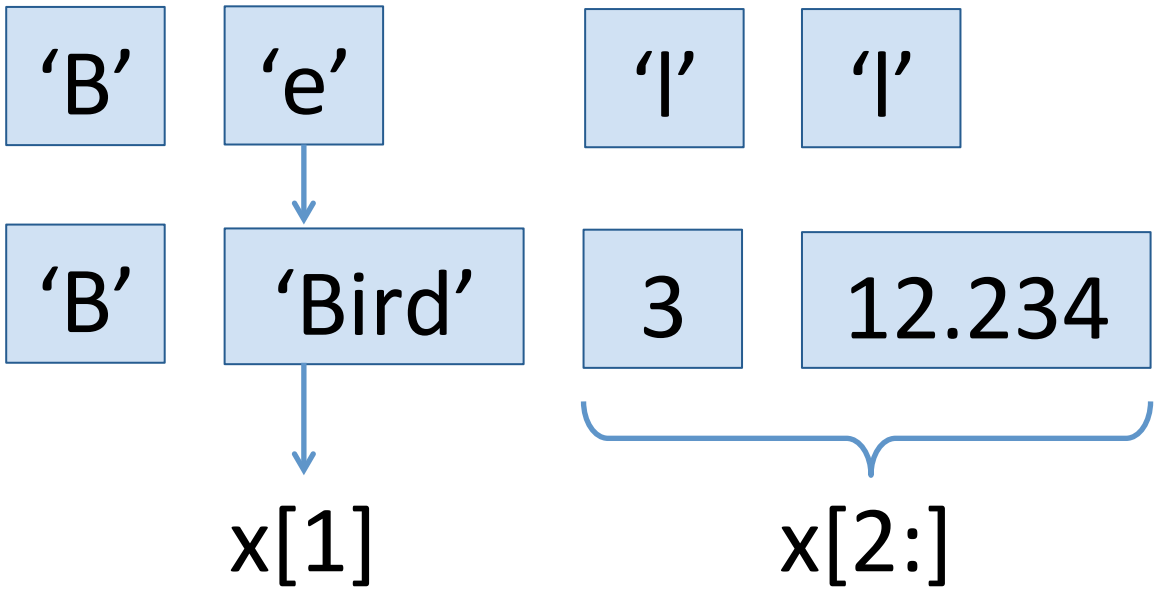
Operations

Concatenation	+
Repetition	*

- Concatenation operates on two lists
- Repetition operates on a list and an integer

Lists are also sequences

- Strings are sequences of characters
- Lists are sequences of any type
  - Even more lists!
- Indexing, slicing, and stepping rules are the same



Mutability

- *Mutability* refers to the ability to change a value once its bound
  - Note the use of *value* and not *variable*
  - This notion specifically refers to the contents of a variable

```
x = 'Hello World'
x = 'Goodbye World'
```

Changing the value of `x` is okay

Lists are mutable. Strings are not.

Lists

```
x = [1, 2, 3]
x[0] = 4
```

Strings

```
x = ‘ello World’
x[0] = ‘H’
```

That’s cool

That’s not cool

Must produce a new string

```
x = ‘Hell World’
x = x[:4] + 'o' + x[4:]
```

List methods

Method	Explanation
`append(x)`	Add `x` to the end of the list
`insert(i, x)`	Insert `x` at position `i`
`remove(x)`	Remove the first instance of `x`
`sort()`	Sort the list
`reverse()`	Reverse the list. Equivalent to using negative step notation
`pop(i)`	Remove, *and return,* the item at the given position. Specifying the position is optional; defaults to -1 (the last element)
`index(x)`	Get the index of the first instance of `x`
`copy()`	*Return* a copy of the list. Equivalent to `x[:]`

- More documentation can be found online

Mutability also applies to the methods

- Recall string *methods*

```
x = 'Hello'
x.lower()
```

  - Like functions, with a bit different syntax
- These methods returned a new string

```
y = x.lower()
```

- This is partially because strings are immutable
- List methods alter the underlying list, in place

```
x = [1, 2, 3]
x.append(4)
```

- The return value of a list method is probably not what you think

```
y = x.append(5)
```

The same, but different: a summary

	Strings	Lists
Types	Characters	Anything
Declaration	`s = 'Hello'`	`s = ['H', 'e', 'l', 'l', 'o']`
Empty	`s = ''`	`s = []`
Mutable	No	Yes

Making a string into a list

- A string can be thought of as a subset of a list
  - They're both sequences
  - Strings have type and mutability restrictions
- Thus, we can turn a string into a list
  1. Cast the string into a list: if you thought `int()` and `float()` were cool, wait until you try `list()`
  2. Split: split is a string method that turns a string into a list where you specify the cut point

Comments (hashtag awesome)

- Comments are a part of good coding
- Serve two purposes
  1. Documentation (formal and no
  2. Code exclusion
- Use comments!
- Use # to create comments in Python
  - Once the parser sees #, it moves on to the next line

Is it funny yet?

```
# Jerome White
# jsw7
# NYUAD CS101 FL 2014
x = 'This little light of mine'
# x = "I'm gonna let it shine"
x[:4].lower().replace('s', 'p')[:1:-1] # 3.14 :)
```