Strings and Console Programs

CS106AP Lecture 7
Roadmap

Day 1!

Programming Basics

The Console

Images

Graphics

Data structures

Midterm

Object-Oriented Programming

Everyday Python

Life after CS106AP!
Today’s questions

How do we build programs that interact with users?

How can we test Python functions?
Today’s topics

1. Review

2. Console programs
   - Console and `input()` function
   - Strings
     - Fundamentals
     - Indexing and slicing
     - Built-in functions

3. Testing
   - Doctests
   - Edge cases

4. What’s next?
Review
For `range()` loops
For loops with `range()`

```python
for i in range(end_index):
    # assumes 0 is the start index
    do_something()

for i in range(start_index, end_index):
    # end_index is not inclusive!
    # recall: range(4,7) -> 4,5,6
    do_something()
```
What’s the difference?

for i in range(5):
    do_something()

i = 0
while i < 5:
    do_something()
    i += 1
What’s the difference?

for i in range(5):
    do_something()

i = 0
while i < 5:
    do_something()
    i += 1

Nothing!
What’s the difference?

```python
for i in range(5):
    do_something()
```

```python
i = 0
while i < 5:
    do_something()
    i += 1
```

Then why use for loops?
What’s the difference?

```
for i in range(5):
    do_something
```

Then why use for loops?
What's the difference?

for i in range(3):
    move()

while front_is_clear():
    move()
What’s the difference?

```python
for i in range(3):
    move()

while front_is_clear():
    move()
```

Use for loops if you know how many times you want to repeat something!
What’s the difference?

for i in range(3):
    move()

Use for loops if you know how many times you want to repeat something!

while front_is_clear():
    move()

Use while loops if you don’t know how many times you want to repeat!
Python functions
Anatomy of a function
Anatomy of a function

input → function(input) → output

parameters/arguments
Anatomy of a function

Function:

input → function(input) → output

**Definition**

**parameter(s)**
One or more variables that your function expects as input
Anatomy of a function

**Definition**

**argument(s)**

The values passed into your function and assigned to its parameter variables.
Anatomy of a function

input → function(input) → output

return value
Anatomy of a function

Definition

**return value**
The value that your function hands back to the “calling” function.
Anatomy of a function
Anatomy of a function

```python
def main():
mid = average(10.6, 7.2)
print(mid)

def average(a, b):
    sum = a + b
    return sum / 2
```

caller
(calling function)

callee
(called function)
def main():
    mid = average(10.6, 7.2)
    print(mid)

def average(a, b):
    sum = a + b
    return sum / 2

Anatomy of a function
def main():
    mid = average(10.6, 7.2)
    print(mid)

def average(a, b):
    sum = a + b
    return sum / 2
Anatomy of a function

def main():
    mid = average(10.6, 7.2)
    print(mid)

def average(a, b):
    sum = a + b
    return sum / 2

These variables only exist inside average()!
def main():
    mid = average(10.6, 7.2)
    print(mid)

def average(a, b):
    sum = a + b
    return sum / 2
def main():
    mid = average(10.6, 7.2)
    print(mid)

def average(a, b):
    sum = a + b
    return sum / 2

Anatomy of a function

This variable only exists inside `main()`!
Common misconceptions: \texttt{print(x)} \textit{vs.} \texttt{return x}

def main():
    get_tip_return(26.50, 6)

def get_tip_return(total_cost, num_people):
    if num_people > THRESHOLD:
        return total_cost * LARGE_PARTY_TIP
    return total_cost * SMALL_PARTY_TIP

def main():
    get_tip_print(26.50, 6)

def get_tip_print(total_cost, num_people):
    if num_people > THRESHOLD:
        print(total_cost * LARGE_PARTY_TIP)
    print(total_cost * SMALL_PARTY_TIP)
Common misconceptions: **print**(*x*) vs. **return** *x*

```python
def main():
    get_tip_return(26.50, 6)

def get_tip_return(total_cost, num_people):
    if num_people > THRESHOLD:
        return total_cost * LARGE_PARTY_TIP
    return total_cost * SMALL_PARTY_TIP
```

```python
def main():
    get_tip_print(26.50, 6)

def get_tip_print(total_cost, num_people):
    if num_people > THRESHOLD:
        print(total_cost * LARGE_PARTY_TIP)
        print(total_cost * SMALL_PARTY_TIP)
```

What does each of these output?
Common misconceptions: \texttt{print(x) \ vs. \ return x}

```python
def main():
    get_tip_return(26.50, 6)

def get_tip_return(total_cost, num_people):
    if num_people > THRESHOLD:
        return total_cost * LARGE_PARTY_TIP
    return total_cost * SMALL_PARTY_TIP

def main():
    get_tip_print(26.50, 6)

def get_tip_print(total_cost, num_people):
    if num_people > THRESHOLD:
        print(total_cost * LARGE_PARTY_TIP)
        print(total_cost * SMALL_PARTY_TIP)
```
Common misconceptions: \texttt{print(x)} vs. \texttt{return x}

```python
def main():
    get_tip_return(26.50, 6)

def get_tip_return(total_cost, num_people):
    if num_people > THRESHOLD:
        return total_cost * LARGE_PARTY_TIP
    return total_cost * SMALL_PARTY_TIP

def main():
    get_tip_print(26.50, 6)

def get_tip_print(total_cost, num_people):
    if num_people > THRESHOLD:
        print(total_cost * LARGE_PARTY_TIP)
        print(total_cost * SMALL_PARTY_TIP)
```

```
Common misconceptions: `print(x)` vs. `return x`

def main():
    tip = get_tip(26.50, 6)
    print(tip)

def get_tip(total_cost, num_people):
    if num_people > THRESHOLD:
        return total_cost * LARGE_PARTY_TIP
    return total_cost * SMALL_PARTY_TIP
Summary

● If you want the calling function to have access to your function’s output, you have to use `return`, not `print()`.
  ○ Make sure to store return values inside the calling function!

● Hitting a `return` line will cause you to immediately exit from a function.
  ○ This is not true with `print()`!

● Only use `print()` for displaying information to the user.
How do we build programs that interact with users?
How do we build programs that interact with users?

Console programs!
**Definition**

**Console program**
A program that solicits input from a user via an interactive terminal (console) and does something interesting with that input.
Two example console programs

This program greets users with a personalized message.
Please enter your name: **Nick**
Please enter your favorite food: **Oreos**
Hello Nick, it's so nice to meet you! Would you like some Oreos to eat?

This program prints squares of numbers.
Please enter a lower bound: **2**
Please enter an upper bound: **11**
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
The square of 6 is 36
The square of 7 is 49
The square of 8 is 64
The square of 9 is 81
The square of 10 is 100
The square of 11 is 121
How do we get information from the user?
How do we get information from the user?

The interactive terminal (console) and the `input()` function!
The console and the `input()` function
The console and the `input()` function

- The console is just another name for the text-output area that we have already seen when using the `print()` function. In addition to displaying text, the console can also solicit text from a user.
The console and the `input()` function

- The console is just another name for the text-output area that we have already seen when using the `print()` function. In addition to displaying text, the console can also solicit text from a user.
- The `input()` function takes in a single parameter, which is a prompt to show to the user.
The console and the `input()` function

- The console is just another name for the text-output area that we have already seen when using the `print()` function. In addition to displaying text, the console can also solicit text from a user.
- The `input()` function takes in a single parameter, which is a prompt to show to the user.
- The function will then wait while the user types in text into the interactive terminal (console).
The console and the \texttt{input()} function

- The console is just another name for the text-output area that we have already seen when using the \texttt{print()} function. In addition to displaying text, the console can also solicit text from a user.
- The \texttt{input()} function takes in a single parameter, which is a prompt to show to the user.
- The function will then wait while the user types in text into the interactive terminal (console).
- After the user submits their answer by hitting the “Enter/Return” key, the function returns the value that the user typed into the console.
The console and the `input()` function

- The console is just another name for the text-output area that we have already seen when using the `print()` function. In addition to displaying text, the console can also solicit text from a user.
- The `input()` function takes in a single parameter, which is a prompt to show to the user.
- The function will then wait while the user types in text into the interactive terminal (console).
- After the user submits their answer by hitting the “Enter/Return” key, the function returns the value that the user typed into the console.

[interpreter demo]
How do we store information from the user?
How do we store information from the user?

Strings!
Strings (a new data type!)
Definition

**string**
A data type that represents a sequence of characters
Definition

string

A data type that represents a sequence of characters

Characters can be letters, digits, symbols (&, !, ~), etc.
Real-life problems involving strings

- Encryption and decryption

encrypted = ‘Jvkpun pz mbu’
decrypted = ‘Coding is fun’

Bonus: What cipher is this?
Real-life problems involving strings

- Encryption and decryption
- DNA Analysis

```
input = 'ATGCGAATGTGC'
output = gene analysis, homology score, etc.
```
Real-life problems involving strings

- Encryption and decryption
- DNA Analysis
- Language translation

input = ‘¿Donde está la biblioteca?’
output = ‘Where is the library?’

*This result cost billions of dollars (adjusted for inflation)*
String fundamentals
String fundamentals

- String literals are any string of characters enclosed in single (‘’) or double quotes (“”)
  - **Note:** We use single quotes in most cases to enclose strings, unless the string itself contains a single quote, in which case we enclose the string with double quotes
String fundamentals

- String literals are any string of characters enclosed in single (‘’) or double quotes (“”)
  - **Note:** We use single quotes in most cases to enclose strings, unless the string itself contains a single quote, in which case we enclose the string with double quotes

    ```
    my_name = 'Nick'
    class_name = 'cs106ap'
    secure_password = 'fg^#kwro!@-lm>'
    sentence = "I don't want that."
    ```
String fundamentals

- String literals are any string of characters enclosed in single (‘’) or double quotes (“”)
- Each character in the string is associated with an index
String fundamentals

- String literals are any string of characters enclosed in single (‘’”) or double quotes (“”)
- Each character in the string is associated with an index

**Definition**

**index**
An integer representing the location of a character in a string
String fundamentals

- String literals are any string of characters enclosed in single (‘‘) or double quotes (“”)
- Each character in the string is associated with an index
  - You can access a character in the sequence via its index using bracket ([ ]) notation
String fundamentals

- String literals are any string of characters enclosed in single (‘’)) or double quotes (“”))
- Each character in the string is associated with an index
  - You can access a character in the sequence via its index using bracket ([ ]) notation

Definition

bracket ([ ]) notation
A character at index i of string s can be accessed with the expression s[i]
String fundamentals

- String literals are any string of characters enclosed in single (‘’’) or double quotes (""")
- Each character in the string is associated with an index
- Strings can be combined with the + operator in a process called concatenation
String fundamentals

- String literals are any string of characters enclosed in single (‘’’) or double quotes (“”)
- Each character in the string is associated with an index
- Strings can be combined with the + operator in a process called concatenation

```python
intro = 'Hello, '
name = 'Nick'
greeting = intro + name  # 'Hello, Nick'
```
String fundamentals

- String literals are any string of characters enclosed in single (‘’’) or double quotes (“”)
- Each character in the string is associated with an index
- Strings can be combined with the + operator in a process called concatenation
- Strings are immutable
String fundamentals

- String literals are any string of characters enclosed in single (‘’’) or double quotes (“”’)
- Each character in the string is associated with an index
- Strings can be combined with the + operator in a process called concatenation
- Strings are immutable
  - Once a string is created, it cannot be modified
String fundamentals

- String literals are any string of characters enclosed in single (‘’ or double quotes (“”)
- Each character in the string is associated with an index
- Strings can be combined with the + operator in a process called concatenation
- Strings are immutable
  - Once a string is created, it cannot be modified
  - To change a string, you must first build a new string via concatenation or a function call, and then re-assign the string variable
String fundamentals

- String literals are any string of characters enclosed in single (’’) or double quotes (“”)
- Each character in the string is associated with an **index**
- Strings can be combined with the + operator in a process called **concatenation**
- Strings are **immutable**
  - Once a string is created, it cannot be modified
  - To change a string, you must first build a new string via concatenation or a function call, and then re-assign the string variable
  - Important consequence: If you pass a string into a function, that function cannot modify the string
String fundamentals

- String literals are any string of characters enclosed in single (‘’') or double quotes (“’”)
- Each character in the string is associated with an index
- Strings can be combined with the + operator in a process called concatenation
- Strings are immutable
Indexing and slicing
Indexing and slicing

- **Length**
  - The *length* of a string is the number of characters it contains.
  - We can use the Python function `len()` to evaluate the length of a string.

  ```python
  len('banana') → 6
  len('') → 0
  len('CS106AP rocks my socks') → 22
  ```
Indexing and slicing

- **Length**
- **Zero-based indexing**
  - The first character of a string exists at index 0
  - The last character of a string exists at index \( \text{len}(s) - 1 \).
  - Index \( \text{len}(s) \) is NOT a valid index.
  - Note: This is a very common paradigm in computer science!

```python
len('Hello!') == 6
```

```
Hello!
```

```
0 1 2 3 4 5
```
Indexing and slicing

- Length
- Zero-based indexing
- Recall: bracket ([ ]) notation
  - The character at index $i$ of string $s$ can be accessed with the expression $s[i]$
Indexing and slicing

- Length
- Zero-based indexing
- Bracket ([])) notation
- Slices and substrings
  - A slice (or substring) of a string is a consecutive block of characters that has been extracted from the original string
  - Whereas indexing uses bracket notation with one number, slicing uses bracket notation with two numbers → \( s[a:b] \)
  - The resulting substring consists of all characters starting from the the first index and going up to, but not including, the second index
Indexing and slicing

[example]
Indexing

\[ s = '{\text{Arthur}}' \]
Indexing

\[ s = 'Arthur' \]
Indexing

\[
s = 'Arthur'
\]

\[
\begin{align*}
  s[0] &= 'A' \\
  s[1] &= 'r' \\
  s[4] &= 'u' \\
  s[6] &= \# \text{ Bad!}
\end{align*}
\]
s = 'Arthur'
Slicing

s = 'Arthur'

Slides courtesy of Sam Redmond, CS41
s = 'Arthur'

s[0:2] == 'Ar'
s = 'Arthur'

s[0:2] == 'Ar'
Slicing

$s = \text{'Arthur'}$

$s[0:2] == \text{'Ar'}$

$s[3:6] == \text{'hur'}$
Slicing

\[ s = \text{\textquotesingle}Arthur\text{\textquotesingle} \]

\[ s[0:2] == \text{\textquotesingle}Ar\text{\textquotesingle} \]
\[ s[3:6] == \text{\textquotesingle}hur\text{\textquotesingle} \]
\[ s[1:4] == \text{\textquotesingle}rth\text{\textquotesingle} \]
Strings

\[ s = 'Arthur' \]

- Implicitly starts at 0
- \[ s[:2] == 'Ar' \]
- \[ s[3:] == 'hur' \]
- Implicitly ends at the end
Strings

\[s = 'Arthur'

- Implicitly starts at 0
- Implicitly ends at the end

\[s[:2] == 'Ar'\]
\[s[3:] == 'hur'\]
Strings

$s = \text{Arthur}$

- Implicitly starts at 0
- $s[2] == \text{'Ar'}$
- $s[3:] == \text{'hur'}$
- Implicitly ends at the end

Slides courtesy of Sam Redmond, CS41
String functions
String functions

- All follow the `noun.verb()` syntax we’ve seen before
  - `noun` is the string variable or literal (unlike the math library)
String functions

- All follow the `noun.verb()` syntax we’ve seen before

- `str.isupper()`, `str.islower()`
  - Return `True` if all the characters in `str` are either uppercase or lowercase, `False` otherwise
String functions

- All follow the `noun.verb()` syntax we’ve seen before

- `str.isupper()`, `str.islower()`

- `str.isalpha()`, `str.isdigit()`
  - Return `True` if all the characters in `str` are either letters (‘a’–‘z’, ‘A’–‘Z’) or digits (‘0’–‘9’),`False` otherwise
String functions

- All follow the `noun.verb()` syntax we’ve seen before

- `str.isupper()`, `str.islower()`

- `str.isalpha()`, `str.isdigit()`

- `str.upper()`, `str.lower()`
  - Returns `str` with all letters converted to uppercase or lowercase respectively
  - The original string remains unchanged
String functions

- All follow the `noun.verb()` syntax we’ve seen before

- `str.isupper()`, `str.islower()`

- `str.isalpha()`, `str.isdigit()`

- `str.upper()`, `str.lower()`
  - Returns `str` with all letters converted to uppercase or lowercase respectively
  - The original string remains unchanged

*Remember: Python strings are immutable!*
String functions

- All follow the `noun.verb()` syntax we’ve seen before

- `str.isupper()`, `str.islower()`

- `str.isalpha()`, `str.isdigit()`

- `str.upper()`, `str.lower()`

[interpreter demo]
Recall: All Python objects have a type!

- Python determines types for you so variables are dynamically-typed.

- We can use the `type()` function to investigate the type of an object.
Recall: All Python objects have a type!

- Python determines types for you so variables are dynamically-typed.
- We can use the `type()` function to investigate the type of an object

How do strings play into all of this?
What does this boolean expression evaluate to?

‘123’ == 123
What does this boolean expression evaluate to?

False!

'123' == 123
Type conversion

- **Important note**: ‘123’ is a **string** and 123 is an **int**
  - Therefore, ‘123’ != 123
  - The two Python objects are of different types and thus can’t be equal!
Type conversion

- **Important note**: ‘123’ is a `str`ing and 123 is an `int`

- In order to convert between data types, we can use built-in Python functions: `str()`, `int()`, `float()`
Type conversion

- **Important note**: ‘123’ is a **string** and 123 is an **int**

- In order to convert between data types, we can use built-in Python functions: `str()`, `int()`, `float()`

  ```python
  int('123') == 123
  float('24.7') == 24.7
  str(12345) == '12345'
  str(20.19) == '20.19'
  ```
Type conversion

- **Important note**: ‘123’ is a **string** and 123 is an **int**

- In order to convert between data types, we can use built-in Python functions: `str()`, `int()`, `float()`

  ```python
  int('123') == 123
  float('24.7') == 24.7
  str(12345) == '12345'
  str(20.19) == '20.19'
  ```

  Now these evaluate to **True!**
Let’s put it all together!
Interactive receipt calculator
[coding demo]
Console program summary

- Use `input(prompt)` to read in information from the user.
  - Make sure to convert the data to the correct type (from the `string` data type)!

- Use `print()` to display information for the user.
  - Make sure to convert the data to the correct type (from the `int/float` data type)

- Use a `while` loop to enable multiple runs of your program.
How can we test Python functions?
How can we test Python functions?

Doctests!
Doctests

- Python has a great testing framework called **doctests**
  - For each function in your program, write doctests that specify an output for a given input
  - You can (and should) have multiple doctests per function
Doctests

- Python has a great testing framework called **doctests**
- PyCharm supports doctests by allowing you to easily run them in the editor
  - Put doctests in function header comments using `>>>`
Doctests

- Python has a great testing framework called **doctests**

- PyCharm supports doctests by allowing you to easily run them in the editor
  - Put doctests in function header comments using `>>>`
    
    ```python
def add(a, b):
    
    >>> add(2, 4)
    6
    >>>
    ...```
Doctests

- Python has a great testing framework called doctests

- PyCharm supports doctests by allowing you to easily run them in the editor
  - Put doctests in function header comments using `>>>`

```python
def add(a, b):
    """
    >>> add(2, 4)
    6
    """
    ...
```

Call the function and specify any arguments if needed.
Doctests

- Python has a great testing framework called **doctests**

- PyCharm supports doctests by allowing you to easily run them in the editor
  - Put doctests in function header comments using `>>>`

```python
def add(a, b):
    """
    >>> add(2, 4)
    6
    """
    ...
```

*Put the expected output directly after the test.*
Testing strategies

- Write tests that cover a wide variety of use cases for your function!

- Consider:
  - Basic use cases
  - Edge cases
Testing strategies

- Write tests that cover a wide variety of use cases for your function!

- Consider:
  - Basic use cases
  - Edge cases

**Definition**

**edge case**

Uses of your function/program that represent extreme situations
Testing strategies

● Write tests that cover a wide variety of use cases for your function!

● Consider:
  ○ Basic use cases
  ○ Edge cases

For example, if your function takes in a string parameter, test what happens if the empty string gets passed in as the argument!

**Definition**

edge case
Uses of your function/program that represent extreme situations
EliminationNation.py
[demo]
Takeaways

- Common pattern: processing all characters in a string

```python
for i in range(len(s)):
    current_char = s[i]
    # Use current_char
```
Takeaways

- Common pattern: processing all characters in a string
- Common pattern: building up a new string

```
new_string = ''
for i in range(len(s)):
    new_string += s[i]
```
Takeaways

● Common pattern: processing all characters in a string

● Common pattern: building up a new string

```python
new_string = ''
for i in range(len(s)):
    if ________:
        new_string += s[i]
```
Takeaways

- Common pattern: processing all characters in a string

- Common pattern: building up a new string

```python
new_string = ''
for i in range(len(s)):
    if __________:
        new_string += s[i]
```

Select only certain characters
Takeaways

● Common pattern: processing all characters in a string

● Common pattern: building up a new string

● Write doctests for every function!
  ○ Cover a range of usage patterns for your function
  ○ Write them before writing the actual function code
  ○ Run them often as you make changes
Takeaways

● Common pattern: processing all characters in a string

● Common pattern: building up a new string

● Write doctests for every function!
  ○ Cover a range of usage patterns for your function
  ○ Write them before writing the actual function code
  ○ Run them often as you make changes
What’s next?
Next week: Images and data structures

- Exploring new ways in which computers store and process structured information
- Making our own version of Photoshop!
- Fun with images and more advanced string processing