YEAH Hours: Assignment 5
Python Programs

Anand Shankar
Slides by Jonathan Kula
October 31, 2019
CS 106AX
Transitioning to Python
Welcome to the wonderful world of Python!

● Instead of being run by your browser (like JavaScript), your Python code is run by the Python interpreter (which you have to install on your computer).
  ○ If you’ve used Python before, we’re asking to install version 3.8 for this course. You can keep any existing versions alongside 3.8.
● We’ll be using the editor “PyCharm” to write code in this course.
● There are instructions on the website, here, to get all set up!
Transitioning to Python: Why?

- What advantages does Python have over JavaScript? Disadvantages?
- Isn’t one programming language enough for a course? Why didn’t we pick one?
- Is it useful to know multiple programming languages?
- What about CS106B/X? Does this prepare us for that?
Transitioning to Python: Why?

What advantages does Python have over JavaScript? Disadvantages?

- It’s less that one is particularly advantageous over the other, and more that they have completely different (and often complimentary) use cases!
- In particular, since JavaScript is run in the web browser and can be “served” (sent) over the internet with website content, it allows for web content to be made dynamic, without having to download anything additional to your computer.
- Meanwhile, Python has developed a great reputation for the large number of well-built scientific computing and machine learning libraries that have been built for it.
  - (That is, Python is one of the most important languages to know if you’re going to be working with data crunching and/or machine learning!).
- Besides this, it has a very expressive and English-like syntax that has a good reputation for being easy to pick up and run with.
Transitioning to Python: Why?

- What advantages does Python have over JavaScript? Disadvantages?
- Isn’t one programming language enough for a course? Why didn’t we pick one?
  - One language can be enough for a course; we’ve seen this plenty of times. What’s the advantage of learning multiple?
  - The reality is that many if not most complex projects, especially projects that make use of the Internet, are often written in two or more separate programming languages.
  - We chose Python as a nice counterpart to JavaScript because of how complementary they are: Python runs on your computer, JavaScript in the browser; Python is a great programming language to quickly script things in; Python can be used as a server that JavaScript code can connect to; and so on and so forth.
Transitioning to Python: Why?

- What advantages does Python have over JavaScript? Disadvantages?
- Isn’t one programming language enough for a course? Why didn’t we pick one?
- Is it useful to know multiple programming languages?
  - Yes! We wouldn’t be teaching you something useless ;)
  - As mentioned earlier, if you find yourself working on a project that uses the internet, you’re likely going to be using at least two programming languages
  - Knowing more programming languages, especially those as complimentary as JavaScript and Python, allows you to cover way more ground with your knowledge.
- What about CS106B/X? Does this prepare us for that?
Transitioning to Python: Why?

- What advantages does Python have over JavaScript? Disadvantages?
- Isn’t one programming language enough for a course? Why didn’t we pick one?
- Is it useful to know multiple programming languages?
- What about CS106B/X? Does this prepare us for that?
  - Yes, better than any other pre-B/X course you could take. Being able to switch between languages is a skill that will be valuable when/if you take B/X. However, because of the languages CS106A/AX are being taught in, we miss a couple concepts that will be important when moving to C++ and other similar languages.
  - The biggest omission is that of strong typing, which means that variables, parameters, and function returns have a declared type, and the actual type of that variable/parameter/function return must match the declared type.
Transitioning to Python: Why?

- What advantages does Python have over JavaScript? Disadvantages?
- Isn’t one programming language enough for a course? Why didn’t we pick one?
- Is it useful to know multiple programming languages?
- What about CS106B/X? Does this prepare us for that?
- Other questions?
Transitioning to Python: How?

Once you’ve gotten your development environment set up, we can start learning new syntax!
Transitioning to Python: Syntax Differences

- Blocks & Indentation
- Variables
- Getting information to the screen
- Comments
- Booleans
- Representing emptiness
- If & Loops
- Operators
- Function calling and definition
Transitioning to Python: Blocks & Indentation

Consider the following JavaScript:

```javascript
if (NUMBER > 0) { // code block begins with {
    doSomethingCool();
    if (ANOTHER_NUMBER < 10) { // blocks can be nested.
        doSomethingElse(NUMBER);
    } else {
        doSomethingElse(ANOTHER_NUMBER);
    }
    doSomethingLast();
} // matching } closes the block
```
Transitioning to Python: Blocks & Indentation

This is the equivalent Python:

```python
if NUMBER > 0:  # blocks start with a colon
    do_something_cool()  # blocks continue with indentation
if ANOTHER_NUMBER < 10:  # blocks can still be nested.
    do_something_else(NUMBER)
else:
    do_something_else(ANOTHER_NUMBER)
do_something_last()
# the block ends when indentation finishes.
```
Transitioning to Python: Blocks & Indentation

Side-by-side:

```python
if (NUMBER > 0) { // code block begins with {
    doSomethingCool();
    if (ANOTHER_NUMBER < 10) { // blocks can be nested.
        doSomethingElse(NUMBER);
    } else {
        doSomethingElse(ANOTHER_NUMBER);
    }
    doSomethingLast();
} // matching } closes the block
```

```python
if NUMBER > 0: # blocks start with a colon :
    do_something_cool() # blocks continue with indentation.
    if ANOTHER_NUMBER < 10: # blocks can still be nested.
        do_something_else(NUMBER)
    else:
        do_something_else(ANOTHER_NUMBER)
    do_something_last()
# the block ends when indentation finishes.
```
Transitioning to Python: Blocks & Indentation

- JavaScript uses curly braces `{ }` to mark the beginning and end of blocks.
- Python uses a colon `:` and indentation instead
- This means bad indentation is a syntax error!
- Also, there aren’t any semicolons :D
Transitioning to Python: Blocks & Indentation

- Note that Python doesn’t allow empty blocks! The above is a syntax error.

```python
if NUMBER > 0:
    # No empty blocks allowed
    # Comments don’t count.
```
Transitioning to Python: Blocks & Indentation

- Note that Python doesn’t allow empty blocks!
- You can use the do-nothing keyword, `pass`, as a placeholder if you want to sketch out a block before implementing it.
Transitioning to Python: Variables

In JavaScript:

```javascript
let variable = "Hello World";
const CONSTANT = 5;
```

In Python:

```python
variable = "Hello World"
CONSTANT = 5
```

Notice the lack of `let` or `const`. 
Transitioning to Python: Printing

In JavaScript, you use `console.log(thingToLog);`

In Python, you use `print(thing_to_log)`
Transitioning to Python: Comments

In JavaScript:

```javascript
// comments begin with //
/* block comments are surrounded by slash-stars */
```

In Python:

```python
# comments begin with a #
""""""block comments are surrounded with triple-quotes""""""
```
Transitioning to Python: Booleans

In JavaScript, you use `true` and `false`

In Python, you use `True` and `False`

Yes, the capitals matter.
Transitioning to Python: Emptiness

In JavaScript, you use `null` and/or `undefined` to express lack of value.

In Python, you use `None` for both.

Yes, the capitals matter.
Transitioning to Python: If & Loops

JavaScript

```javascript
if (ANOTHER_NUMBER < 10) {
    doSomethingElse(NUMBER);
} else {
    doSomethingElse(ANOTHER_NUMBER);
}
```

Python

```python
if ANOTHER_NUMBER < 10:
    doSomethingElse(NUMBER)
else:
    doSomethingElse(ANOTHER_NUMBER)
```
Transitioning to Python: If & Loops

JavaScript

```javascript
if (ANOTHER_NUMBER < 10) {
  doSomethingElse(NUMBER);
} else {
  doSomethingElse(ANOTHER_NUMBER);
}
```

Python

```python
if ANOTHER_NUMBER < 10:
  do_something_else(NUMBER)
else:
  do_something_else(ANOTHER_NUMBER)
```

- No parentheses
Transitioning to Python: If & Loops

JavaScript

```javascript
if (ANOTHER_NUMBER < 10) {
    doSomethingElse(NUMBER);
} else if (ANOTHER_NUMBER === 10) {
    doSomethingElse(ANOTHER_NUMBER);
}
```

Python

```python
if ANOTHER_NUMBER < 10:
    do_something_else(NUMBER)
elif ANOTHER_NUMBER == 10:
    do_something_else(ANOTHER_NUMBER)
```
Transitioning to Python: If & Loops

- While loops function in exactly the same way:

```python
# prints 0, then 1, ..., then 9, then stops
i = 0
while i < 10:
    print(i)
    i += 1

let i = 0;
while (i < 10) {
    console.log(i);
    i ++;
}
```
Transitioning to Python: If & Loops

- For loops, however, work very differently:

```python
# prints 0, then 1, ..., then 9, then stops
for i in range(10):
    print(i)
```

```javascript
for (let i = 0; i < 10; i++) {
    console.log(i);
}
```
Transitioning to Python: If & Loops

- In JavaScript, a for loop is just a nice place to move a bunch of parts of a while loop together:

```javascript
let i = 0;
while (i < 10) {
    console.log(i);
    i++;
}
```

```python
for (let i = 0; i < 10; i++) {
    console.log(i);
}
```
In JavaScript, a for loop is just a nice place to move a bunch of parts of a while loop together:

```javascript
let i = 0;
while (i < 10) {
    console.log(i);
    i++;
}

for (let i = 0; i < 10; i++) {
    console.log(i);
}
```
Transitioning to Python: If & Loops

- In JavaScript, a for loop is just a nice place to move a bunch of parts of a while loop together:

```javascript
let i = 0;
while (i < 10) {
    console.log(i);
    i++;
}
```

```python
for (let i = 0; i < 10; i++) {
    console.log(i);
}
```
Transitioning to Python: If & Loops

- In JavaScript, a for loop is just a nice place to move a bunch of parts of a while loop together:

```javascript
let i = 0;
while (i < 10) {
  console.log(i);
  i++;
}
```

```javascript
for (let i = 0; i < 10; i++) {
  console.log(i);
}
```
Transitioning to Python: If & Loops

- Meanwhile, in Python, for loops loop over a collection of values.

```python
# prints 0, then 1, ..., then 9, then stops
for i in range(10):
    print(i)
```
Transitioning to Python: If & Loops

- Meanwhile, in Python, for loops loop over a collection of values.

```python
# prints 0, then 1, ..., then 9, then stops
for i in range(10):
    print(i)
```

- In this case, `range(10)` is a collection of numbers between 0 and 9. (the value entered is an exclusive end point).
Transitioning to Python: If & Loops

- Meanwhile, in Python, for loops loop over a collection of values.

```python
# prints 0, then 1, ..., then 9, then stops
for i in range(10):
    print(i)
    i = 100  # for resets this next loop
```

- In this case, `range(10)` is a collection of numbers between 0 and 9. (the value entered is an exclusive end point).
- Because of this, the above modification won’t actually affect the output at all!
Transitioning to Python: If & Loops

- Meanwhile, in Python, for loops loop over a *collection of values*.

```python
# prints Hello, then There, then stops
for i in "Hello", "There":
    print(i)
```

- In this case, `"Hello", "There"` is a *list of strings*. (note we call arrays “lists” in Python)
- Note that because `for` goes over a collection of values, we can use any valid collection!
Transitioning to Python: If & Loops

- Meanwhile, in Python, for loops loop over a collection of values.

```python
# prints Hello, then There, then stops
for string in ["Hello", "There"]: print(string)
```

- In this case, ["Hello", "There"] is a list of strings.
- Note that because `for` goes over a collection of values, we can use any valid collection!
Transitioning to Python: If & Loops

- Meanwhile, in Python, for loops loop over a collection of values.

```python
# prints P, then y, ..., then n, then stops
for i in "Python":
    print(i)
```

- In this case, "Python" is a string, which is interpreted as a collection of characters.
- Note that because for goes over a collection of values, we can use any valid collection!
Transitioning to Python: If & Loops

- Meanwhile, in Python, for loops loop over a collection of values.

```python
# prints P, then y, ..., then n, then stops
for char in "Python":
    print(char)
```

- In this case, "Python" is a string, which is interpreted as a collection of characters.
- Note that because `for` goes over a collection of values, we can use any valid collection!
Transitioning to Python: If & Loops

- Meanwhile, in Python, for loops loop over a collection of values.

```
# prints 0, then 1, ..., then 5, then stops
for i in range(len("Python")):
    print(i)
```

- In this case, we’re using `range` again. If we want to go over the indices of a string or list, JavaScript style, we can use `len` to get the length.
- Note that because `for` goes over a collection of values, we can use any valid collection!
Transitioning to Python: Operators (Arith)

Arithmetic operators are the same:

```
i + 1  # addition
i - 1  # subtraction
i * 1  # multiplication
i / 1  # division
i % 1  # modulo
```

...but Python gives us a couple more!

```
i ** 2  # power (i to the power of 2)
i // 2  # floor division (in JS: Math.floor(i / 2))
```
Transitioning to Python: Operators (Boolean)

Boolean operators are different:

Javascript:

\[
\begin{align*}
&a \& b \\
&a || b \\
&!a
\end{align*}
\]

Python:

\[
\begin{align*}
&a \text{ and } b \\
&a \text{ or } b \\
&\text{not } a
\end{align*}
\]
Transitioning to Python: Operators (Comp)

Comparison operators are mostly the same:

```
i < 1  # less than
i <= 1 # less than or equal
i > 1  # more than
i >= 1 # more than or equal
i == 1 # equal (NOTE: don’t use ===)
i != 1 # not equal (NOTE: don’t use !==)
```

...but Python gives us another useful operator!

```
"c" in "collection" # membership (True)
"c" in ["a", "bc"]  # False; checks only one level down.
```
Python also has some handy-dandy syntax for “between”:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>0 &lt; i &lt; 1</code></td>
<td>between 0 and 1 (exclusive)</td>
</tr>
<tr>
<td><code>0 &lt;= i &lt;= 1</code></td>
<td>between 0 and 1 (inclusive)</td>
</tr>
<tr>
<td><code>0 &lt;= i &lt; 1</code></td>
<td>between 0 and 1 (inclusive of 0 only)</td>
</tr>
<tr>
<td><code>0 &lt; i &lt;= 1</code></td>
<td>between 0 and 1 (inclusive of 1 only)</td>
</tr>
</tbody>
</table>
Transitioning to Python: Function definition

In JavaScript:

```javascript
function myFunction(param1, param2) {
    // your implementation here
}
```

In Python:

```python
def my_function(param1, param2):
    pass  # your implementation here
```
Transitioning to Python: Function definition

In JavaScript:

```javascript
function myFunction(param1, param2) {
    // your implementation here
}
```

In Python (using JavaScript-style naming – still works):

```python
def myFunction(param1, param2):
    pass  # your implementation here
```
Transitioning to Python: Function definition

In JavaScript:

```javascript
function myFunction(param1, param2) {
    // your implementation here
}
```

In Python (using JavaScript-style naming – still works):

```python
def myFunction(param1, param2):
    pass  # your implementation here
```

Function calling works the exact same way in both: e.g. `myFunction(1, "Hi")`
Transitioning to Python: Scope

- Everyone’s favorite, ~scope~!

Consider the following JavaScript:

```javascript
if (true) {
    let variable = 1;
}
console.log(variable);
```
Transitioning to Python: Scope

- Everyone’s favorite, ~scope~!

Consider the following JavaScript:

```javascript
if (true) {
    let variable = 1;
}
console.log(variable); // ERROR! variable not defined
```
Transitioning to Python: Scope

- Everyone’s favorite, ~scope~!

Now consider the following Python:

```python
if True:
    variable = 1

print(variable)
```
Transitioning to Python: Scope

- Everyone’s favorite, ~scope~!

Now consider the following Python:

```python
if True:
    variable = 1

print(variable)  # Prints 1
```
Transitioning to Python: Scope

● Everyone’s favorite, ~scope~!

Now consider the following Python:

```python
if True:
    variable = 1

print(variable)  # Prints 1
```

This is a very dangerous affordance!
Transitioning to Python: Scope

- Everyone’s favorite, ~scope~!

Now consider the following Python:

```python
if False:
    variable = 1

print(variable)  # ERROR! variable not defined!
```

This is a **very dangerous affordance**!
Transitioning to Python: Scope

- Everyone’s favorite, ~scope~!

Now consider the following Python:

```python
if False:
    variable = 1
else:
    variable = 0
print(variable)  # variable is guaranteed to be defined here.
```

This is a very dangerous affordance! Use it only when you know for sure your variable will be defined when you use it!
Transitioning to Python: Style Differences

In JavaScript, it’s the industry convention to use camelCase, which is what we’ve been telling you all year in IGs.

However, Python conventionally uses snake_case for its variable and function naming.
Transitioning to Python: Style Differences

In JavaScript, it’s the industry convention to use camelCase, which is what we’ve been telling you all year in IGs.

However, Python conventionally uses snake_case for its variable and function naming. (I have no idea if the pun is intentional or not)
Transitioning to Python: Style Differences

In JavaScript, it’s the industry convention to use camelCase, which is what we’ve been telling you all year in IGs.

However, Python conventionally uses snake_case for its variable and function naming. (Whatever the case may be, this is the actual industry standard)
Transitioning to Python: Style Differences

In JavaScript, it’s the industry convention to use camelCase, which is what we’ve been telling you all year in IGs.

However, Python conventionally uses snake_case for its variable and function naming. (Whatever the case may be, this is the actual industry standard)

Because we’ve been teaching you camelCase all quarter and will continue to in the slides, we will accept either as valid style, but please choose one and be consistent.
Assignment 5
Assignment 5: Logistics

- Due on November 6th
- If you do extensions, submit a regular and extended version of your files
- ...yup, that’s all!
Assignment 5: Python Programs

- Two problems meant to help you dive right into Python while still exercising your developing programming expertise

- Problem 1: Random Sentence Generator

- Problem 2: Reassemble

- Although we have separate problems, the problems themselves are not separated into milestones.
  - They are mostly pre-decomposed into parts, though! :}
Part I: Random Sentence Generator
Random Sentence Generator: Theory

- You’ll be using something called a context-free grammar to generate random sentences that adhere to a particular set of rules. We abbreviate this as “CFG”
- A CFG is a type of formal language, used in mathematics, computer science, and linguistics.
- A CFG defines rules; a certain sentence is considered in that language if it conforms to the rules of the CFG
- You’ll be generating sentences that conform to a given grammar!
Random Sentence Generator: CFGs

- To the right is a basic CFG.

```
<start>
1
The <object> <verb> tonight.

<object>
3
waves
big yellow flowers
slugs

<verb>
3
sigh <adverb>
portend like <object>
die <adverb>

<adverb>
2
warily
grumpily
```
Random Sentence Generator: CFGs

- To the right is a basic CFG.
- A CFG contains several definitions.
Random Sentence Generator: CFGs

- To the right is a basic CFG.
- A CFG contains several definitions.
- Definitions start with a nonterminal.
Random Sentence Generator: CFGs

- To the right is a basic CFG.
- A CFG contains several definitions.
- Definitions start with a **nonterminal.**
  - Nonterminals are *placeholders*. They will be later expanded into one of their **productions.**
Random Sentence Generator: CFGs

- To the right is a basic CFG.
- A CFG contains several definitions.
- Definitions start with a nonterminal.
  - Nonterminals are *placeholders*. They will be later expanded into one of their *productions*.
  - Productions contain space-separated *tokens*. 

```text
<start>
1
The <object> <verb> tonight.

<object>
3
waves
big yellow flowers
slugs

<verb>
3
sigh <adverb>
portend like <object>
die <adverb>

<adverb>
2
warily
grumpily
```
Random Sentence Generator: CFGs

- To the right is a basic CFG.
- A CFG contains several definitions.
- Definitions start with a **nonterminal**.
  - Nonterminals are *placeholders*. They will be later expanded into one of their *productions*.
  - Productions contain space-separated tokens.
  - Tokens may be **terminal** or **nonterminal**.
Random Sentence Generator: CFGs

- To the right is a basic CFG.
- A CFG contains several definitions.
- Definitions start with a **nonterminal**.
  - Nonterminals are *placeholders*. They will be later expanded into one of their *productions*.
  - Productions contain space-separated tokens.
  - Tokens may be **terminal** or **nonterminal**.
  - If a production contains nonterminals, it will have to eventually be expanded.
Random Sentence Generator: CFGs

- To the right is a basic CFG.
- A CFG contains several definitions.
- Definitions start with a nonterminal.
  - Nonterminals are *placeholders*. They will be later expanded into one of their *productions*.
  - Productions contain space-separated tokens.
  - Tokens may be terminal or nonterminal.
  - If a *production contains nonterminals*, it will have to eventually be expanded.
Random Sentence Generator: CFGs

- To the right is a basic CFG.
- A CFG contains several definitions.
- Definitions start with a nonterminal.
  - Nonterminals are *placeholders*. They will be later expanded into one of their *productions*.
  - Productions contain space-separated tokens.
  - Tokens may be terminal or nonterminal.
  - If a production contains nonterminals, it will have to eventually be expanded.
  - Immediately after the starting nonterminal, there is a *number* that says how many *productions* the definition has.
To the right is a basic CFG.

A CFG contains several definitions.

Definitions start with a nonterminal.

- Nonterminals are placeholders. They will be later expanded into one of their productions.
- Productions contain space-separated tokens.
- Tokens may be terminal or nonterminal.
- If a production contains nonterminals, it will have to eventually be expanded.
- Immediately after the starting nonterminal, there is a number that says how many productions the definition has.
- The empty string is a valid production.
Random Sentence Generator: CFGs

- To the right is a basic CFG.
- A CFG contains several definitions.
- Definitions start with a nonterminal.
  - Nonterminals are *placeholders*. They will be later expanded into one of their *productions*.
  - Productions contain space-separated tokens.
  - Tokens may be terminal or nonterminal.
  - If a production contains nonterminals, it will have to eventually be expanded.
  - Immediately after the starting nonterminal, there is a number that says how many productions the definition has.
  - The empty string is a valid production.

- You may assume our CFGs are correctly formatted, i.e. you don't have to worry about error-checking.
Random Sentence Generator: Generating

- Let’s generate a sentence.

```html
<start>
1
The <object> <verb> tonight.

<object>
3
waves
big yellow flowers
slugs

<verb>
3
sigh <adverb>
portend like <object>
die <adverb>

<adverb>
2
warily
grumpily
```
Random Sentence Generator: Generating

- Let’s generate a sentence.
- We start with the `<start>` definition

```plaintext
<start>
1
The <object> <verb> tonight.

<object>
3
waves
big yellow flowers
slugs

<verb>
3
sigh <adverb>
portend like <object>
die <adverb>

<adverb>
2
warily
grumpily
```
Let’s generate a sentence.

We start with the <start> definition

Choose a random entry from its productions
Let’s generate a sentence.
We start with the `<start>` definition
Choose a random entry from its productions
Hey look, there’s only one!

`<start>`

The `<object> <verb> tonight.`

<obj> 3
waves
big yellow flowers
slugs

<verb> 3
sigh <adv>
portend like <obj>
die <adv>

<adv> 2
warily
grumpily
Random Sentence Generator: Generating

- Let’s generate a sentence.
- We start with the <start> definition
- Choose a random entry from its productions
- Hey look, there’s only one!

The <object> <verb> tonight.
Random Sentence Generator: Generating

- Let’s generate a sentence.
- We start with the <start> definition
- Choose a random entry from its productions
- Does our expansion have nonterminals?

The <object> <verb> tonight.
Random Sentence Generator: Generating

- Let’s generate a sentence.
- We start with the `<start>` definition
- Choose a random entry from its productions
- Does our expansion have nonterminals? Yes!

The `<object>` `<verb>` `tonight`. 
Random Sentence Generator: Generating

- Let’s generate a sentence.
- We start with the <start> definition
- Choose a random entry from its productions
- Let’s expand nonterminals.

The <object> <verb> tonight.
Random Sentence Generator: Generating

- Let’s generate a sentence.
- We start with the <start> definition
- Choose a random entry from its productions
- Let’s expand nonterminals. Same process!

The <object> <verb> tonight.
Random Sentence Generator: Generating

- Let’s generate a sentence.
- We start with the <start> definition
- Choose a random entry from its productions
- Let’s expand nonterminals. Same process!
- We’ll start with <object>.

The <object> <verb> tonight.

- waves
- big yellow flowers
- slugs

- sigh <adverb>
- portend like <object>
- die <adverb>

- warily
- grumpily
Random Sentence Generator: Generating

- Let’s generate a sentence.
- We start with the `<start>` definition
- Choose a random entry from its productions
- Let’s expand nonterminals. Same process!
- We’ll start with `<object>`.
- Randomly pick a production.

The `<object>` `<verb>` tonight.
Random Sentence Generator: Generating

- Let’s generate a sentence.
- We start with the <start> definition
- Choose a random entry from its productions
- Let’s expand nonterminals. Same process!
- We’ll start with <object>.
- Randomly pick a production. I rolled a 3.

The <object> <verb> tonight.

```
<start>
1
The <object> <verb> tonight.

<object>
3
waves
big yellow flowers
slugs

<verb>
3
sigh <adverb>
portend like <object>
die <adverb>

<adverb>
2
warily
grumpily
```
Random Sentence Generator: Generating

- Let’s generate a sentence.
- We start with the `<start>` definition.
- Choose a random entry from its productions.
- Let’s expand nonterminals. Same process!
- We’ll start with `<object>`.
- Randomly pick a production. I rolled a 3.
- Now substitute the nonterminal for the production we picked.

The `<object>` `<verb>` tonight.
Let’s generate a sentence.

We start with the `<start>` definition.

Choose a random entry from its productions.

Let’s expand nonterminals. Same process!

We’ll start with `<object>`.

Randomly pick a production. I rolled a 3.

Now substitute the nonterminal for the production we picked.

The slugs `<verb>` tonight.
Let’s generate a sentence.
We start with the `<start>` definition
Choose a random entry from its productions
Let’s expand nonterminals. Same process!
We’ll start with `<object>`.
Randomly pick a production. I rolled a 3.
Now substitute the nonterminal for the production we picked.
Now repeat.

The **slugs** `<verb>` **tonight**.
Let’s generate a sentence.

We start with the <start> definition

Choose a random entry from its productions

Let’s expand nonterminals. Same process!

We’ll continue with <verb>

The slugs <verb> tonight.
Random Sentence Generator: Generating

- Let’s generate a sentence.
- We start with the `<start>` definition.
- Choose a random entry from its productions.
- Let’s expand nonterminals. Same process!
- We’ll continue with `<verb>`.

```
<start>
1
The <object> <verb> tonight.

<object>
3
waves
big yellow flowers
slugs

<verb>
3
sigh <adverb>
portend like <object>
die <adverb>

<adverb>
2
warily
grumpily
```

The slugs `<verb>` tonight.
Random Sentence Generator: Generating

- Let’s generate a sentence.
- We start with the `<start>` definition.
- Choose a random entry from its productions.
- Let’s expand nonterminals. Same process!
- We’ll continue with `<verb>`.
- I’ll randomly pick a production.

The slugs `<verb>` tonight.
Random Sentence Generator: Generating

- Let’s generate a sentence.
- We start with the <start> definition.
- Choose a random entry from its productions.
- Let’s expand nonterminals. Same process!
- We’ll continue with <verb>.
- I’ll randomly pick a production. Rolled a 2.

The slugs <verb> tonight.
Let’s generate a sentence.

We start with the `<start>` definition.

Choose a random entry from its productions.

Let’s expand nonterminals. Same process!

We’ll continue with `<verb>`.

I’ll randomly pick a production. Rolled a 2.

Substitute!

The slugs `<verb>` tonight.
Random Sentence Generator: Generating

- Let’s generate a sentence.
- We start with the <start> definition
- Choose a random entry from its productions
- Let’s expand nonterminals. Same process!
- We’ll continue with <verb>.
- I’ll randomly pick a production. Rolled a 2.
- Substitute!

```
<start>
1
The <object> <verb> tonight.

<object>
3
waves
big yellow flowers
slugs

<verb>
3
sigh <adverb>

portend like <object>
die <adverb>

<adverb>
2
warily
grumpily
```
Let’s generate a sentence.
We start with the <start> definition
Choose a random entry from its productions
Let’s expand nonterminals. Same process!
We’ll continue with <verb>.
I’ll randomly pick a production. Rolled a 2.
Substitute!
Repeat!

The slugs portend like <object> tonight.

The <object> <verb> tonight.

<object>
3
waves
big yellow flowers
slugs
<verb>
3
sigh <adverb>
portend like <object>
die <adverb>

<adverb>
2
warily
grumpily

95
Random Sentence Generator: Generating

- Let’s generate a sentence.
- We start with the `<start>` definition
- Choose a random entry from its productions
- Let’s expand nonterminals. Same process!
- Notice we have a new nonterminal!

The slugs portend like <object> tonight.
Random Sentence Generator: Generating

- Let’s generate a sentence.
- We start with the <start> definition
- Choose a random entry from its productions
- Let’s expand nonterminals. Same process!
- We’ll continue with <object>.

The slugs portend like <object> tonight.
Random Sentence Generator: Generating

- Let’s generate a sentence.
- We start with the <start> definition
- Choose a random entry from its productions
- Let’s expand nonterminals. Same process!
- We’ll continue with <object>.
- Randomly pick a production.

The slugs portend like <object> tonight.
Let's generate a sentence.

We start with the `<start>` definition.

Choose a random entry from its productions.

Let’s expand nonterminals. Same process!

We’ll continue with `<object>`.

Randomly pick a production. Rolled a 2.

The slugs portend like `<object>` tonight.
Random Sentence Generator: Generating

- Let’s generate a sentence.
- We start with the <start> definition
- Choose a random entry from its productions
- Let’s expand nonterminals. Same process!
- We’ll continue with <object>.
- Randomly pick a production. Rolled a 2.
- Substitute!

The slugs portend like **big yellow flowers** tonight.
Random Sentence Generator: Generating

- Let’s generate a sentence.
- We start with the `<start>` definition
- Choose a random entry from its productions
- Let’s expand nonterminals. Same process!
- We’ll continue with `<object>`.
- Randomly pick a production. Rolled a 2.
- Substitute!
- And now we’re out of non-terminals!

The slugs portend like big yellow flowers tonight.
Let’s generate a sentence.

We start with the <start> definition.

Choose a random entry from its productions.

Let’s expand nonterminals. Same process!

We’ll continue with <object>.

Randomly pick a production. Rolled a 2.

Substitute!

And now we’re out of non-terminals!

That means we’re done :)

The slugs portend like big yellow flowers tonight.
Random Sentence Generator: Generating

The slugs portend like big yellow flowers tonight.
The slugs portend like big yellow flowers tonight. Beautiful.
The slugs portend like big yellow flowers tonight.

10/10 would poem again.
Random Sentence Generator: Structure

GenerateRandomSentences.py is already decomposed for you!
# File: GenerateRandomSentences.py
# ----------------------------------
# This file exports a program that reads in a grammar file and
# then prints three randomly generated sentences

from filechooser import chooseInputFile
from random import choice

def readGrammar(filename):
    return {}  # replace this with your own implementation

def generateRandomSentence(grammar):
    return "[placeholder for a random sentence]"  # replace this with your own implementation

def GenerateRandomSentences():
    filename = chooseInputFile("grammars")
    grammar = readGrammar(filename)
    for i in range(3):
        print(generateRandomSentence(grammar))

if __name__ == "__main__":
    GenerateRandomSentences()
Random Sentence Generator: Structure

Let’s figure out a good data structure for a grammar.
Random Sentence Generator: Structure

Let’s figure out a good data structure for a grammar.

We can start with thinking about definitions.
Let’s figure out a good data structure for a grammar.

We can start with thinking about definitions. These are the main building blocks of the grammar.
Let’s figure out a good data structure for a grammar.

We can start with thinking about definitions. These are the main building blocks of the grammar.

How can we identify a definition? How do we look up a definition?
Let’s figure out a good data structure for a grammar.

We can start with thinking about definitions. These are the main building blocks of the grammar.

How can we identify a definition?
How do we look up a definition?
With the non-terminal!
Let’s figure out a good data structure for a grammar.

We notice that each non-terminal uniquely identifies a definition.
Random Sentence Generator: Structure

Let’s figure out a good data structure for a grammar.

We notice that each non-terminal **uniquely** identifies a definition.
Let’s figure out a good data structure for a grammar.

We notice that each non-terminal uniquely identifies a definition.

This is a key -> value situation!
Random Sentence Generator: Structure

Let’s nail down what a definition is!
Random Sentence Generator: Structure

Let’s take a look at <object>
Random Sentence Generator: Structure

Let’s take a look at <object>

What information does it have?
Random Sentence Generator: Structure

Let’s take a look at <object>

What information does it have?
Random Sentence Generator: Structure

Let’s take a look at `<object>`

What information does it have?

1) The non-terminal
2) The number
3) The productions
Random Sentence Generator: Structure

Let’s take a look at `<object>`

What information does it have?

1) The non-terminal
2) The number
3) The productions

What information do we need to keep around for when we’re generating sentences?
Random Sentence Generator: Structure

Let’s take a look at <object>

What information does it have?

1) The non-terminal
2) The number (no need; once we store our productions we already know how many there are)
3) The productions

What information do we need to keep around for when we’re generating sentences?

<start>
1
The <object> <verb> tonight.

<object>
3
waves
big yellow flowers
slugs

<verb>
3
sigh <adverb>
portend like <object>
die <adverb>

<adverb>
2
warily
grumpily
Random Sentence Generator: Structure

Let’s take a look at <object>

What information does it have?

1) The non-terminal (depending on how you store your grammar, you may not need this either!)

2) The number (no need; once we store our productions we already know how many there are)

3) The productions

What information do we need to keep around for when we’re generating sentences?
Random Sentence Generator: Structure

Let’s take a look at `<object>`

What information does it have?

1) The non-terminal
2) The productions

How will we represent the productions?
Random Sentence Generator: File Reading

We’re working backwards here :)

Now that we know the purpose for which we’re creating a data structure, and we know the data structure we’ll use, it’s finally time to actually populate that structure. We’ll do that by reading in a grammar file.
Random Sentence Generator: File Reading

We’re already familiar with the file!

<start>
1
The <object> <verb> tonight.

<object>
3
waves
big yellow flowers
slugs

<verb>
3
sigh <adverb>
portend like <object>
die <adverb>

<adverb>
2
warily
grumpily
Random Sentence Generator: File Reading

We’re already familiar with the file!

We have a bunch of definitions...
We’re already familiar with the file!

We have a bunch of definitions...

...and each definition has three pieces of information.
Random Sentence Generator: File Reading

We’re already familiar with the file!

We have a bunch of definitions...

...and each definition has three pieces of information.

The key,
Random Sentence Generator: File Reading

We’re already familiar with the file!

We have a bunch of definitions...

...and each definition has three pieces of information.

The key, the number of productions,
Random Sentence Generator: File Reading

We’re already familiar with the file!

We have a bunch of definitions...

...and each definition has three pieces of information.

The key, the number of productions, and the productions themselves.
Random Sentence Generator: File Reading

We’re working with lines when we’re reading this file.
Random Sentence Generator: File Reading

We’re working with lines when we’re reading this file.

Each line has a different meaning.

That means that splitting the file up into lines is gonna be the way to go!
We can open the file like so:

```
with open(filename) as f:
```

There’s two ways we can get all the lines:

```
for line in f:
```

and

```
lines = f.readlines()
```

Which should we use?
Random Sentence Generator: File Reading

We can open the file like so:

```python
with open(filename) as f:
```

There's two ways we can get all the lines:

```python
for line in f:
```

and

```python
lines = f.readlines()
```

Which should we use? Why?
Random Sentence Generator

Once you’ve read everything into the data structure, all that’s left to do is implement the sentence generator as described previously. Then you’re done!
Part II: Reassemble
Reassemble: Theory

- You’ll be taking fragments of a message, and reassembling them back into the message!

- You’ll do this by finding the longest overlap between all available fragments and merging those two together, then repeating until you’re done with all the fragments!

- You’ll also have to read the fragments in from the files we give you.
Reassemble: File Reading

Fragment files look like the above.
Fragment files look like the above. Here's another example:

{did Peter Piper pick?}{icked.
If Peter P}{ers did Peter Piper pic}{ckled peppers d}{led peppers did Peter Pi}{ peppers
How }{peppers
How many pick}{per pick?
}{rs.
A peck of pickled peppers Peter Piper picked.
If Peter ){peppers.
A peck of pic}{many pickled pepp}{Peter Piper picked a peck of pickled peppers
}{pickled peppers.
A peck}{Peter Piper picked a peck of pickled peppers.
A pe}
Reassemble: File Reading

Fragment files look like the above. Here’s another example:

{did Peter Piper pick?}{icked.
If Peter P}{ers did Peter Piper pic}{ckled peppers d}{led peppers did Peter Pi}{ peppers
How ){peppers
How many pick){per pick?
}{rs.
A peck of pickled peppers Peter Piper picked.
If Peter ){peppers.
A peck of pic}{many pickled pepp}{Peter Piper picked a peck of pickled peppers
}{pickled peppers.
A peck}{Peter Piper picked a peck of pickled peppers.
A pe}

Notice that fragments can span lines!
Reassemble: File Reading

Since fragments can span lines, we won’t read the file line by line.

Instead, we can do:

```python
with open(filename) as f:
    text = f.read()
```

This will read the entire file in as a single string. From here, you can take each fragment and put it in a list!
Reassemble: Reassembly

Now that we have all our fragments, let’s consider how to reassemble them.

We’ll start with an example!
Reassemble: Reassembly

Now that we have all our fragments, let’s consider how to reassemble them.

We’ll start with an example!

This will eventually reassemble into the string “all is well that ends well”

Let’s explore how we get there.
Reassemble: Reassembly

We know that we’re trying to find the longest overlap between two fragments.

Here are our pairings:

all is well + ell that en
all is well + hat end
all is well + t ends well
ell that en + hat end
ell that en + t ends well
hat end + t ends well
Reassemble: Reassembly

We know that we’re trying to find the **longest overlap** between two fragments.

Here are our pairings (note that the order doesn’t matter):

- all is well + ell that en
- all is well + hat end
- all is well + t ends well
- ell that en + hat end
- ell that en + t ends well
- hat end + t ends well
Reassemble: Reassembly

We know that we’re trying to find the longest overlap between two fragments.

Here are our pairings:

all is well + ell that en (all is well that en) (3)
all is well + hat end (no overlap) (0)
all is well + t ends well (no overlap) (0)
ell that en + hat end (ell hat end) (6)
ell that en + t ends well (t ends well that en) (3)
hat end + t ends well (hat ends well) (5)
Reassemble: Reassembly

We know that we’re trying to find the longest overlap between two fragments.

Here are our pairings:

all is well + ell that en (all is well that en) (3)
all is well + hat end (no overlap) (0)
all is well + t ends well (no overlap) (0)

ell that en + hat end (ell that end) (6)
ell that en + t ends well (t ends well that en) (3)
hat end + t ends well (hat ends well) (5)
Reassemble: Reassembly

We know that we’re trying to find the **longest overlap** between two fragments.

Here are our pairings:

- all is well + ell that end
- all is well + t ends well
- ell that end + t ends well
Reassemble: Reassembly

We know that we’re trying to find the longest overlap between two fragments.

Here are our pairings:

all is well + ell that end (all is well that end) (3)
all is well + t ends well (no overlap) (0)
ell that end + t ends well (ell that ends well) (5)
We know that we’re trying to find the longest overlap between two fragments.

Here are our pairings:

all is well + ell that end (all is well that end) (3)
all is well + t ends well (no overlap) (0)
ell that end + t ends well (ell that ends well) (5)
Reassemble: Reassembly

We know that we’re trying to find the **longest overlap** between two fragments.

Here are our pairings:

all is well + ell that ends well
Reassemble: Reassembly

We know that we’re trying to find the **longest overlap** between two fragments.

Here are our pairings:

```
all is well
ell that ends well
```

all is well + ell that ends well (all is well that ends well) (3)
We know that we’re trying to find the *longest overlap* between two fragments.

Here are our pairings:

```
all is well + ell that ends well (all is well that ends well) (3)
```
Reassemble: Reassembly

We know that we’re trying to find the *longest overlap* between two fragments.
Reassemble: Reassembly

We’ve run out of fragments to combine, so we’re good!

all is well that ends well
In terms of programming this, you’ll be working with your list of fragments.

Here are a couple methods you may find yourself using:

- `list.pop(i)`  # removes the element at index i and returns it
- `list.remove(v)`  # removes the first item from the list that matches v.

You’ll likely have to use a double for loop, but can you figure out how to make sure it doesn’t test the same pairing twice?
Reassemble: Reassembly

In terms of programming this, you’ll be working with your list of fragments.

Here are a couple methods you may find yourself using:

- `list.pop(i)` # removes the element at index i and returns it
- `list.remove(v)` # removes the first item from the list that matches v.

You’ll likely have to use a double for loop, but can you figure out how to make sure it doesn’t test the same pairing twice?

We also strongly recommend implementing a function that checks the length of the overlap between two strings. This will make your job of finding the maximum length much, much easier and cleaner.
Reassemble: Edge Cases: No Match

Consider the following example.
Reassemble: Edge Cases: No Match

Consider the following example.

Here are our pairings:

abc + xyz
Reassemble: Edge Cases: No Match

Consider the following example.

Here are our pairings:

abc + xyz (no match) (0)
Reassemble: Edge Cases: No Match

Consider the following example.

Here are our pairings:

abc + xyz (no match) (0)

In this case, we can just concatenate them.
Reassemble: Edge Cases: No Match

Consider the following example.

Here are our pairings:

abc + xyz (no match) (0)

In this case, we can just concatenate them.
Reassemble: Edge Cases: No Match

Consider the following example.

Done!

In this case, we can just concatenate them.
Reassemble: Edge Cases: No Match

Consider the following example.

Done!

In this case, we can just concatenate them.

Order doesn’t matter; this would also be a valid reassembly of those fragments.
Consider the following example.

Here are our pairings:

abc + cde
abc + efg
cde + efg
Reassemble: Edge Cases: Multiple Best

Consider the following example.

Here are our pairings:

- abc + cde (abc\text{cde}) (1)
- abc + efg (no match) (0)
- cde + efg (cde\text{efg}) (1)
Consider the following example.

Here are our pairings:

- $\text{abc} + \text{cde (abcde)} (1)$
- $\text{abc} + \text{efg (no match)} (0)$
- $\text{cde} + \text{efg (cdefg)} (1)$

In this case, when you have multiple best matches, just choose one of those matches. It doesn’t matter which.
Reassemble: Edge Cases: Multiple Best

Consider the following example.

Here are our pairings:

\[ \text{abcde} + \text{efg} \ (\text{abcde} \text{efg}) \ (1) \]
Reassemble: Edge Cases: Multiple Best

Consider the following example.

Done!
Reassemble: Strategy

- Work with the smaller fragment files first. The bigger ones may take a very long time to run the first time around, and it’s harder to verify that you did them correctly.
- Do the file reading first and ensure that it’s correct before moving on to reassembly.
- To make your program run faster, try and ensure you’re not checking the same two strings twice. (e.g. don’t check abc + cde then cde + abc – you’ll be doing duplicate work!)
Questions?