Style and (more) Control Flow

CS106AP Lecture 4
Roadmap

Programming Basics

The Console

Images

Data structures

Midterm

Graphics

Object-Oriented Programming

Everyday Python

Life after CS106AP!

Day 1!
Today’s questions

What do we mean by coding with “good style”?

How does control flow allow us to create more complex programs?
Today’s topics

1. Review: HurdlesKarel

2. Coding style tactics

3. Control flow 2.0
   and/or
   Conditionals (if-else, if-elif-else)
   Example: NinjaKarel

4. What’s next?
Review
Program structure
The structure of a **program**

```python
import ...

def main():
    ...

if __name__ == '__main__':
    ...
```

**Definition**

**computer program**
A set of instructions for the computer – an “app.” These instructions are also called lines of “source code.”
not
A note about `not`

- We can use `not` in front of a boolean expression to negate it!

```python
if not boolean expression:
    # Do something if expression is not True
```
A note about **not**

- We can use **not** in front of a boolean expression to negate it!

```python
if not front_is_clear():
    turn_around()
```
Top-down decomposition
Top-down decomposition

- “Divide-and-conquer”
  - Break the problem down into smaller parts
  - Ask: What are the steps that make up the larger problem? What tasks are repeated and might make good functions?
Top-down decomposition

● “Divide-and-conquer”

● Plan out your milestones (functions) first before writing them:
  ○ What are the pre-conditions and post-conditions for each function?
  ○ Which functions will call which?
Top-down decomposition

- “Divide-and-conquer”

- Plan out your milestones (functions) first before writing them

- Use the “blackbox model” for functions you’re not working on
  - When working on a particular function, assume that the others exist and already do what you want
Top-down decomposition

- “Divide-and-conquer”
- Plan out your milestones (functions) first before writing them
- Use the “blackbox model” for functions you’re not working on
Steps

1. Plan your milestones (functions).

2. Write all of the function definitions and their pre-conditions and post-conditions in the function comments.

3. Work on one function at a time: write its code, test it, and debug as necessary before moving on to the next milestone (function).

4. Make sure your program works on all possible scenarios (worlds)!
Today we’re going to create...
ATHLETE KAREL!
Another example: HurdlesKarel
Hurdles Karel

begin

end
Karel will always start with a hurdle directly to its right.
HurdlesKarel

Four moves, two types
Hurdles

Follow wall on right
Hurdles

Follow wall until blocked
Steps

1. Plan your milestones (functions).

2. Write **all of the function definitions** and their pre-conditions and post-conditions in the function comments.

3. Work on **one function at a time**: write its code, test it, and debug as necessary before moving on to the next milestone (function).

4. Make sure your program works on all possible scenarios (worlds)!
HurdlesKarel
[demo]
Hurdles

Karel

begin

end
What do we mean by coding with “good style”?
In this class...

● Style pointers will be called out:

**Style note**

decomposition
Having multiple small functions makes your program easier to read and understand
In this class...

- Style pointers will be called out.

- Each assignment has a tab with tips in the course Style Guide.

![WhiteSpace and Indentation](image)
In this class...

- Style pointers will be called out.
- Each assignment has a tab with tips in the course Style Guide.
- We will use PEP8!
A note about PEP 8

● PEP - Python Enhancement Proposal

● **PEP 8** is an official standard for Python code formatting

● All the code we show you will follow PEP 8, and PyCharm also enforces it by default!
  ○ You should therefore follow this by default in your code.

● In real-world projects, some are super strict about following PEP 8.
What we’ve learned so far

● Have descriptive function names
  ○ Use verbs
  ○ Separate words with underscores (this is called “snake case”)
What we’ve learned so far

● Have descriptive function names

● Decompose your programs
  ○ Use short functions to add descriptive names or add basic control flow lines to an existing command (e.g. `move_safely()`)
  ○ Create functions that reduce repeated code and isolate specific sub-problems.
What we’ve learned so far

● Have descriptive function names

● Decompose your programs

● Use consistent spacing (4 spaces per level of indentation!)
  ○ This is also a common source of bugs
  ○ Python uses indentation to understand your code (what’s inside the function/loop/if statement)
What we’ve learned so far

- Have descriptive function names
- Decompose your programs
- Use consistent spacing (4 spaces per level of indentation!)

- Comment your code as you write each function
  - Include overall comments for every function
  - Use inline comments for complex blocks of code within functions
What we’ve learned so far

- Have descriptive function names
- Decompose your programs
- Use consistent spacing (4 spaces per level of indentation!)
- Comment your code as you write each function
How does control flow enable more complex programs?
Combining boolean expressions
Definition

boolean expression
A code snippet that evaluates to True or False
**Definition**

**boolean expression**

A code snippet that evaluates to True or False

*Used as the conditions in if statements and while loops!*
Karel functions that evaluate to boolean expressions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>front_is_clear()</code></td>
<td>Is there no wall in front of Karel?</td>
</tr>
<tr>
<td><code>left_is_clear()</code></td>
<td>Is there no wall to Karel’s left?</td>
</tr>
<tr>
<td><code>right_is_clear()</code></td>
<td>Is there no wall to Karel’s right?</td>
</tr>
<tr>
<td><code>on_beeper()</code></td>
<td>Is there a beeper on the corner where Karel is standing?</td>
</tr>
<tr>
<td><code>facing_north()</code></td>
<td>Is Karel facing north?</td>
</tr>
<tr>
<td><code>facing_south()</code></td>
<td>Is Karel facing south?</td>
</tr>
<tr>
<td><code>facing_east()</code></td>
<td>Is Karel facing east?</td>
</tr>
<tr>
<td><code>facing_west()</code></td>
<td>Is Karel facing west?</td>
</tr>
</tbody>
</table>
Python uses the words **not, and, & or** to combine boolean values.
not, and, & or

- Python uses the words not, and, & or to combine boolean values
  - not boolean_expression
    - Inverts True/False
When is the condition true?

<table>
<thead>
<tr>
<th>a</th>
<th>not a</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>
not, and, & or

- Python uses the words **not**, **and**, & **or** to combine boolean values
  - **not** boolean_expression
    - Inverts True/False
  - boolean_expression_a **and** boolean_expression_b
    - *If both sides are true, the entire condition is true.*
When is the condition true?

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>a and b</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---------</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
not, and, & or

- Python uses the words **not, and, & or** to combine boolean values
  - **not boolean_expression**
    - Inverts True/False
  - **boolean_expression_a and boolean_expression_b**
    - If both sides are true, the entire condition is true.
  - **boolean_expression_a or boolean_expression_b**
    - If either side is true, the entire condition is true.
When is the condition true?

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>a or b</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---------</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
not, and, & or

- Python uses the words **not**, **and**, **&** and **or** to combine boolean values
  - **not** boolean_expression
    *Inverts True/False*
  - boolean_expression_a **and** boolean_expression_b
    *If both sides are true, the entire condition is true.*
  - boolean_expression_a **or** boolean_expression_b
    *If either side is true, the entire condition is true.*
Conditionals
(if-else)
If statements

if ___________:
    # Do something
If statements

```python
if Condition:
    # Do something
```

(Condition: any boolean expression)
If statements

```
if ___________:
    # Do something
```

Condition

True    False
If statements

```
if ___________:
    # Do something
```

Condition

True  False
If statements

if ___________:
    # Do something

Condition

True False

X
If-else statements

```python
if ____________:
    # Do something

else:
    # Do something different
```
If-else statements

```
if ___________:
    # Do something
else:
    # Do something different
```
If-else statements

```
if ___________:
  # Do something
else:
  # Do something different
```

Condition

- True
- False
If-else statements

Condition

if ___________:
# Do something
else:
# Do something different
If-else statements

```python
if ___________
    # Do something
else:
    # Do something different
```
If-else statements

if light is red:
    Stop
else:
    Go
If-else statements

if light is red:
    Stop
else:
    Go

What about yellow lights?
Conditionals
(if-elif-else)
If-else statements

```python
if ___________:
    # Do something

else:
    # Do something different
```
If-elif-else statements

if ___________:
    # Do something

elif ___________:
    # Do something different

delse:
    # Do another different thing
If-elif-else statements

You can have multiple elif blocks.

```python
if __________:
    # Do something

elif __________:
    # Do something different

else:
    # Do another different thing
```
If-elif-else statements

```python
if ____________:
    # Do something

elif ____________:
    # Do something different

else:
    # Do another different thing
```

The else is optional!
If-elif-else statements

```python
if ___________:
    # Do something

elif ___________:
    # Do something different

else:
    # Do another different thing
```
If-elif-else statements

if light is red:
    Stop

elif light is yellow:
    Slow down

else:
    Go
If-elif-else statements

- Can have more than one elif block
  - But too many can get messy

- Else is optional
  - You can think of this like a default option

- Put the conditions in priority order!
  - Ask: Which condition do I want to check first?
If-elif-else statements

- Can have more than one elif block
  - But too many can get messy

- Else is optional
  - You can think of this like a default option

- Put the conditions in priority order!
  - Ask: Which condition do I want to check first?

Why not just use multiple if statements?
What’s the difference?

if light is red:
    Stop

elif light is yellow:
    Slow down

else:
    Go

if light is red:
    Stop

if light is yellow:
    Slow down

if light is green:
    Go
What’s the difference?

if light is red:
    Stop
elif light is yellow:
    Slow down
else:
    Go

if light is red:
    Stop
if light is yellow:
    Slow down
if light is green:
    Go

If the light is red, we shouldn’t need to check anything else!
Let’s put it all together!
NinjaKarel
A beeper means you should jump to the left.
No beeper means you should jump to the right.
NinjaKarel [demo]
What’s next?
You have everything you need for Assignment 1!