B Pvote source code

The following pages present the source code of Pvote, consisting of seven modules:

- `main.py`
- `Ballot.py`
- `verifier.py`
- `Navigator.py`
- `Audio.py`
- `Video.py`
- `Printer.py`

Each line of code is numbered and printed in monospaced type.

```
42          self.bindings = get_list(stream, Binding)
```

Defining occurrences of classes, methods, and functions appear in bold.

```
127 def get_enum(stream, cardinality):
```

Lines marked with a triangle are entry points into a module, called from other modules. Functions and methods without a triangle are called only from within the same module.

```
▷48     def press(self, key):
```

The code is broken into sections, with explanatory text in grey preceding each section.

Explanatory text looks like this.

Reviewers' comments, from the Pvote security review, are marked with bullets and shown in grey italic text after the section to which they refer.

- *Reviewers' notes look like this.*

main.py

This is the main Pvote program. It initializes the other software components with the provided ballot definition file and then processes incoming Pygame events in a non-terminating loop.

```
1 import Ballot, verifier, Audio, Video, Printer, Navigator, pygame
```

These two constants are the type IDs of user-defined events. An `AUDIO_DONE` event signals that an audio clip has finished playing. A `TIMER_DONE` event signals that a timed delay has elapsed.

```
2 AUDIO_DONE = pygame.USEREVENT
3 TIMER_DONE = pygame.USEREVENT + 1
```

Reviewers suggested that all constants be moved into a separate module; thus, for example, both `main.py` and `Audio.py` would refer to the same `AUDIO_DONE` constant instead of redundantly defining it in both files.

The following lines load the ballot definition, verify it, and then instantiate the other parts of Pvote with their corresponding sections of the ballot definition.

```
4 ballot = Ballot.Ballot(open("ballot"))
5 verifier.verify(ballot)
6 audio = Audio.Audio(ballot.audio)
7 video = Video.Video(ballot.video)
8 printer = Printer.Printer(ballot.text)
9 navigator = Navigator.Navigator(ballot.model, audio, video, printer)
```

This is the main event loop. The loop begins by updating the display to match the framebuffer in memory, so that any display changes made during the last iteration appear onscreen. The loop never exits.

```
10 while 1:
11     pygame.display.update()
```

On each iteration, one event is retrieved from Pygame's event queue. A timeout is scheduled before waiting for the event, so that if no events occur in `timeout_ms` milliseconds, a `TIMER_DONE` event will be posted. This timeout is then cancelled so that a timer event cannot occur while other processing is taking place.

```
12     pygame.time.set_timer(TIMER_DONE, ballot.model.timeout_ms)
13     event = pygame.event.wait()
14     pygame.time.set_timer(TIMER_DONE, 0)
```

Keypresses are handled by the navigator's **`press()`** method. Touches on the touchscreen are handled by looking for a corresponding target; if one is found, the event is handled by the navigator's **`touch()`** method.

```
15     if event.type == pygame.KEYDOWN:
16         navigator.press(event.key)
17     if event.type == pygame.MOUSEBUTTONDOWN:
18         [x, y] = event.pos
19         target_i = video.locate(x, y)
20         if target_i != None:
21             navigator.touch(target_i)
```

The audio driver schedules an `AUDIO_DONE` event to be posted whenever an audio clip finishes playing. Upon receipt of such an event, the audio driver's **`next()`** method is called so that any audio clips waiting to be played next can start playing.

```
22     if event.type == AUDIO_DONE:
23         audio.next()
```

If a `TIMER_DONE` event was received, that means there has been no user activity for `timeout_ms` milliseconds. It also means that no `AUDIO_DONE` event has occurred for `timeout_ms` milliseconds, which means that either the audio is silent or that a clip has been playing for longer than `timeout_ms` milliseconds. If the `playing` flag on the audio driver is zero, that means the timeout period has elapsed since the last user input occurred or last audio clip finished.

```
24     if event.type == TIMER_DONE and not audio.playing:
25         navigator.timeout()
```

Ballot.py

The `Ballot` module defines the ballot definition data structure. The main program instantiates a **`Ballot`** object to deserialize the ballot data from a file stream and construct the ballot definition data structure. All the other classes in this module represent parts of the ballot definition; each one deserializes its contents from the stream passed to its constructor.

```
 1 import sha

 2 class Ballot:
 3     def __init__(self, stream):
 4         assert stream.read(8) == "Pvote\x00\x01\x00"
 5         [self.stream, self.sha] = [stream, sha.sha()]
```

In order to produce a SHA-1 hash of all the ballot data, the **`Ballot`** object passes `self` as the stream object to the other constructors. Its `read` method allows it to proxy for the original stream, allowing it to incorporate all the data into the hash as it passes through. After all four parts of the ballot definition have been loaded, the last 20 bytes of the stream are checked to ensure they match the hash.

```
 6         self.model = Model(self)
 7         self.text = Text(self)
 8         self.audio = Audio(self)
 9         self.video = Video(self)
10         assert self.sha.digest() == stream.read(20)

11     def read(self, length):
12         data = self.stream.read(length)
13         self.sha.update(data)
14         return data
```

*Reviewers suggested that the **`read()`** method would make more sense if moved into a separate object playing the role of the stream proxy, instead of using the **`Ballot`** itself as the stream proxy. This change would also prevent the sub-objects from having access to the incompletely constructed **`Ballot`** object during construction.*

Each remaining class loads its contents from the stream in a constructor that parallels its data structure. These constructors instantiate other classes to read single components from the stream, call **`get_list()`** to read a variable-length list of components from the stream, or call **`get_int()`**, **`get_enum()`**, or **`get_str()`** to deserialize primitive data types from the stream.

```
15 class Model:
16     def __init__(self, stream):
17         self.groups = get_list(stream, Group)
18         self.pages = get_list(stream, Page)
19         self.timeout_ms = get_int(stream, 0)

20 class Group:
21     def __init__(self, stream):
22         self.max_sels = get_int(stream, 0)
23         self.max_chars = get_int(stream, 0)
24         self.option_clips = get_int(stream, 0)
25         self.options = get_list(stream, Option)
```

```
class Option:
    def __init__(self, stream):
        self.sprite_i = get_int(stream, 0)
        self.clip_i = get_int(stream, 0)
        self.writein_group_i = get_int(stream, 1)

class Page:
    def __init__(self, stream):
        self.bindings = get_list(stream, Binding)
        self.states = get_list(stream, State)
        self.option_areas = get_list(stream, OptionArea)
        self.counter_areas = get_list(stream, CounterArea)
        self.review_areas = get_list(stream, ReviewArea)

class State:
    def __init__(self, stream):
        self.sprite_i = get_int(stream, 0)
        self.segments = get_list(stream, Segment)
        self.bindings = get_list(stream, Binding)
        self.timeout_segments = get_list(stream, Segment)
        self.timeout_page_i = get_int(stream, 1)
        self.timeout_state_i = get_int(stream, 0)

class OptionArea:
    def __init__(self, stream):
        self.group_i = get_int(stream, 0)
        self.option_i = get_int(stream, 0)

class CounterArea:
    def __init__(self, stream):
        self.group_i = get_int(stream, 0)
        self.sprite_i = get_int(stream, 0)

class ReviewArea:
    def __init__(self, stream):
        self.group_i = get_int(stream, 0)
        self.cursor_sprite_i = get_int(stream, 1)

class Binding:
    def __init__(self, stream):
        self.key = get_int(stream, 1)
        self.target_i = get_int(stream, 1)
        self.conditions = get_list(stream, Condition)
        self.steps = get_list(stream, Step)
        self.segments = get_list(stream, Segment)
        self.next_page_i = get_int(stream, 1)
        self.next_state_i = get_int(stream, 0)

class Condition:
    def __init__(self, stream):
        self.predicate = get_enum(stream, 3)
        self.group_i = get_int(stream, 1)
        self.option_i = get_int(stream, 0)
        self.invert = get_enum(stream, 2)

class Step:
    def __init__(self, stream):
        self.op = get_enum(stream, 5)
        self.group_i = get_int(stream, 1)
        self.option_i = get_int(stream, 0)
```

```
class Segment:
    def __init__(self, stream):
        self.conditions = get_list(stream, Condition)
        self.type = get_enum(stream, 5)
        self.clip_i = get_int(stream, 0)
        self.group_i = get_int(stream, 1)
        self.option_i = get_int(stream, 0)

class Text:
    def __init__(self, stream):
        self.groups = get_list(stream, TextGroup)

class TextGroup:
    def __init__(self, stream):
        self.name = get_str(stream)
        self.writein = get_enum(stream, 2)
        self.options = get_list(stream, get_str)

class Audio:
    def __init__(self, stream):
        self.sample_rate = get_int(stream, 0)
        self.clips = get_list(stream, Clip)
```

The **Clip** type contains the waveform data for an audio clip, which resides in a single Python string. In a serialized ballot definition, the number of samples is stored preceding the audio data. Since each sample is a 16-bit value, the number of bytes to read is twice the number of samples.

```
class Clip:
    def __init__(self, stream):
        self.samples = stream.read(get_int(stream, 0)*2)

class Video:
    def __init__(self, stream):
        self.width = get_int(stream, 0)
        self.height = get_int(stream, 0)
        self.layouts = get_list(stream, Layout)
        self.sprites = get_list(stream, Image)

class Layout:
    def __init__(self, stream):
        self.screen = Image(stream)
        self.targets = get_list(stream, Rect)
        self.slots = get_list(stream, Rect)
```

An **Image** object contains the pixel data for an image, which resides in a single Python string. In serialized form, the image's width and height are stored preceding the pixel data, which contains three bytes per pixel (one byte each for the red, green, and blue components).

```
class Image:
    def __init__(self, stream):
        self.width = get_int(stream, 0)
        self.height = get_int(stream, 0)
        self.pixels = stream.read(self.width*self.height*3)

class Rect:
    def __init__(self, stream):
        self.left = get_int(stream, 0)
        self.top = get_int(stream, 0)
        self.width = get_int(stream, 0)
        self.height = get_int(stream, 0)
```

The **get_int()** function reads an unsigned 4-byte integer from the stream. The `allow_none` argument is a flag specifying whether the returned value can be `None`, which is represented by the sequence `"\xff\xff\xff\xff"`. This function ensures that the data meets the constraints given in the assurance document—namely, that the value is between 0 and $2^{31} - 1$ inclusive, or `None` only for fields that allow it.

```
122  def get_int(stream, allow_none):
123      [a, b, c, d] = list(stream.read(4))
124      if ord(a) < 128:
125          return ord(a)*16777216 + ord(b)*65536 + ord(c)*256 + ord(d)
126      assert allow_none and a + b + c + d == "\xff\xff\xff\xff"
```

- *Reviewers suggested that it would be clearer to have two separate methods (for reading an integer and reading an integer-or-None) instead of using **get_int()** for both purposes.*

- *Reviewers agreed that there should be an explicit `return None` statement to show that `None` is the intended return value.*

The **get_enum()** function reads an enumerated type from the stream, which is represented the same way as an integer. The second argument gives the cardinality of the enumeration, which is used to ensure the validity of the returned value.

```
127 def get_enum(stream, cardinality):
128     value = get_int(stream, 0)
129     assert value < cardinality
130     return value
```

- *Reviewers suggested that it would be clearer to have two separate methods for reading Boolean values and enumerated values, instead of using `get_enum(stream, 2)` to read Boolean values.*

The **get_str()** function reads a string from the stream, which is represented as a sequence of bytes prefixed by the length as a 4-byte integer. This function checks that all the characters in the string fall in the printable ASCII range, so they will print out in a predictable way. The tilde character (number 126) is specifically excluded to avoid any ambiguity in the printed output, because the tilde is used as a delimiter.

```
131 def get_str(stream):
132     str = stream.read(get_int(stream, 0))
133     for ch in list(str):
134         assert 32 <= ord(ch) <= 125
135     return str
```

- *Reviewers suggested that the condition in line 134 would be easier to understand if it were written `isprint(ch) and ch != '~'`.*

The **get_list()** function reads a variable-length list of data structures from the stream, all of a particular given class. In Python (and Pthin), classes are first-class objects and can be passed as arguments. In serialized form, the list is preceded by a 4-byte integer indicating how many elements to read.

```
136 def get_list(stream, Class):
137     return [Class(stream) for i in range(get_int(stream, 0))]
```

verifier.py

The `verifier` module contains only one entry point, **`verify()`**, whose responsibility is to abort the program if the ballot definition is not well-formed. The intention is that, if execution continues after a call to **`verify()`**, it should never abort thereafter—that is: (a) **`verify()`** checks all the assumptions about the ballot definition upon which the rest of Pvote relies; and (b) the contents of the ballot definition data structures are never changed after **`verify()`** is called.

```
▷  1  def verify(ballot):
   2      [groups, sprites] = [ballot.model.groups, ballot.video.sprites]
```

`option_sizes` contains one list corresponding to each group; it will collect all the sprites for the options in that group and all the slots in which such options could be pasted (in option areas and review areas). `char_sizes` also contains one list for each group; it will collect all the sprites for characters corresponding to write-in options in the group, as well as all the slots in which such characters could be pasted (in review areas). These lists will later be checked to ensure that the sizes of all sprites match the sizes of all the slots into which they could be pasted.

```
   3      option_sizes = [[] for group in groups]
   4      char_sizes = [[] for group in groups]
```

The following lines ensure that the parallel arrays have matching size. It also makes sure that they are also nonempty; for example, the navigator assumes that there is at least one page when it starts up with a transition to page 0.

```
   5      assert len(ballot.model.groups) == len(ballot.text.groups) > 0
   6      assert len(ballot.model.pages) == len(ballot.video.layouts) > 0
```

For each page, the list of bindings are checked. Each page also has to have at least one state.

```
   7      for [page_i, page] in enumerate(ballot.model.pages):
   8          layout = ballot.video.layouts[page_i]

   9          for binding in page.bindings:
  10              verify_binding(ballot, page, binding)
  11          assert len(page.states) > 0
```

For each state, the segments and bindings are checked. The sprite is checked to make sure it exactly fills its slot, and the timeout transition is also checked for validity.

```
  12          for [state_i, state] in enumerate(page.states):
  13              verify_size(sprites[state.sprite_i], layout.slots[state_i])
  14              verify_segments(ballot, page, state.segments)
  15              for binding in state.bindings:
  16                  verify_binding(ballot, page, binding)
  17              verify_segments(ballot, page, state.timeout_segments)
  18              verify_goto(ballot, state.timeout_page_i, state.timeout_state_i)
  19          slot_i = len(page.states)
```

Each option area is checked for a valid option reference, and the option slots are gathered into the appropriate array for later size checking.

```
        for area in page.option_areas:
            verify_option_ref(ballot, page, area)
            option_sizes[area.group_i].append(layout.slots[slot_i])
            slot_i = slot_i + 1
```

For each counter area, all the possible sprites that could be pasted are checked to ensure they exactly fill the slot.

```
        for area in page.counter_areas:
            for i in range(groups[area.group_i].max_sels + 1):
                verify_size(sprites[area.sprite_i + i], layout.slots[slot_i])
            slot_i = slot_i + 1
```

For each review area, the slots for options and characters are gathered into the appropriate array for later size checking. If there is a cursor sprite, its size is expected to match the option slots as well.

```
        for area in page.review_areas:
            for i in range(groups[area.group_i].max_sels):
                option_sizes[area.group_i].append(layout.slots[slot_i])
                slot_i = slot_i + 1
                for j in range(groups[area.group_i].max_chars):
                    char_sizes[area.group_i].append(layout.slots[slot_i])
                    slot_i = slot_i + 1
            if area.cursor_sprite_i != None:
                option_sizes[area.group_i].append(sprites[area.cursor_sprite_i])
```

The sprites for all the options and characters are gathered into the appropriate arrays. The audio clip indices for the options are ensured to be within range. For write-in options, the number of allowed write-in characters in the parent group is checked to ensure it matches the number of allowed selections in the write-in group; thus, all the write-in options in a group are required to accept the same number of characters. Write-in groups are not themselves allowed to contain write-ins.

```
    for [group_i, group] in enumerate(groups):
        for option in group.options:
            option_sizes[group_i].append(sprites[option.sprite_i])
            option_sizes[group_i].append(sprites[option.sprite_i + 1])
            assert group.option_clips > 0
            ballot.audio.clips[option.clip_i + group.option_clips - 1]
            if option.writein_group_i != None:
                writein_group = groups[option.writein_group_i]
                assert writein_group.max_chars == 0
                assert writein_group.max_sels == group.max_chars > 0
                for option in writein_group.options:
                    char_sizes[group_i].append(sprites[option.sprite_i])
```

The sprites and slots that have been collected for each group are now checked to ensure they all have matching sizes.

```
        for object in option_sizes[group_i]:
            verify_size(object, option_sizes[group_i][0])
        for object in char_sizes[group_i]:
            verify_size(object, char_sizes[group_i][0])
```

The text section is checked to ensure that every option has a name, and ensure that the group names and option names have reasonable lengths that will print properly.

```
53      for [group_i, group] in enumerate(ballot.text.groups):
54          assert len(group.name) <= 50
55          assert len(group.options) == len(groups[group_i].options)
56          for option in group.options:
57              assert len(option) <= 50
```

Every audio clip is checked to ensure that it has nonzero length. There is no Pvote code that relies on this property; Pygame has the an unfortunate limitation that the audio system will abort if asked to play a zero-length sound.

```
58      for clip in ballot.audio.clips:
59          assert len(clip.samples) > 0
```

Finally, the video section is checked. The background images must match the screen size, all the slots and targets must fit entirely onscreen, and the image data for each sprite must match the sprite's claimed dimensions.

```
60      assert ballot.video.width*ballot.video.height > 0
61      for layout in ballot.video.layouts:
62          verify_size(layout.screen, ballot.video)
63          for rect in layout.targets + layout.slots:
64              assert rect.left + rect.width <= ballot.video.width
65              assert rect.top + rect.height <= ballot.video.height
66      for sprite in ballot.video.sprites:
67          assert len(sprite.pixels) == sprite.width*sprite.height*3 > 0
```

The **verify_binding()** function checks that a binding is well-formed by inspecting each of its parts: its list of conditions, its list of steps, its list of audio segments, and its transition.

```
68  def verify_binding(ballot, page, binding):
69      for condition in binding.conditions:
70          verify_option_ref(ballot, page, condition)
71      for step in binding.steps:
72          verify_option_ref(ballot, page, step)
73      verify_segments(ballot, page, binding.segments)
74      verify_goto(ballot, binding.next_page_i, binding.next_state_i)
```

The **verify_goto()** function checks that the page index and state index for a transition are within range. None is an allowed value for the page index.

```
75  def verify_goto(ballot, page_i, state_i):
76      if page_i != None:
77          ballot.model.pages[page_i].states[state_i]
```

The **verify_segments()** function checks that a list of segments is well-formed. It inspects each segment's list of conditions and, based on the segment type, ensures that all the possible corresponding indices of audio clips are within range.

```
78 def verify_segments(ballot, page, segments):
79     for segment in segments:
80         for condition in segment.conditions:
81             verify_option_ref(ballot, page, condition)
82         ballot.audio.clips[segment.clip_i]
83         if segment.type in [1, 2, 3, 4]:
84             group = verify_option_ref(ballot, page, segment)
85             if segment.type in [1, 2]:
86                 assert segment.clip_i < group.option_clips
87             if segment.type in [3, 4]:
88                 ballot.audio.clips[segment.clip_i + group.max_sels]
```

Reviewers wanted to see meaningfully named constants here for the enumerated values. They recommended that all the enumerated value constants should be pulled out into a separate module—thus, for example, the above code and the navigator code would refer to the same set of SG_* constants.

The **verify_option_ref()** function checks the validity of an (indirect or direct) option reference in a condition, step, or segment—all of these types have a `group_i` field and an `option_i` field. If the `group_i` field is `None`, then `option_i` must be the index of a valid option area on the current page. Otherwise, `group_i` and `option_i` must be valid group and option indices respectively. The group object is returned as a convenience for **verify_segments()**, which uses the group object for other checks.

```
89 def verify_option_ref(ballot, page, object):
90     if object.group_i == None:
91         area = page.option_areas[object.option_i]
92         return ballot.model.groups[area.group_i]
93     ballot.model.groups[object.group_i].options[object.option_i]
94     return ballot.model.groups[object.group_i]
```

The **verify_size()** function ensures that two objects (sprites or slots) have the same dimensions.

```
95 def verify_size(a, b):
96     assert a.width == b.width and a.height == b.height
```

Navigator.py

The first three lines set up constants corresponding to the three enumerated types in the ballot model definition: `OP_*` for step types, `SG_*` for audio segment types, and `PR_*` for predicates in conditions.

```
1  [OP_ADD, OP_REMOVE, OP_APPEND, OP_POP, OP_CLEAR] = range(5)
2  [SG_CLIP, SG_OPTION, SG_LIST_SELS, SG_COUNT_SELS, SG_MAX_SELS] = range(5)
3  [PR_GROUP_EMPTY, PR_GROUP_FULL, PR_OPTION_SELECTED] = range(3)
```

The navigator is initialized with access to the ballot model data structure, audio driver, video driver, and printing module. It saves these references locally, initializes an empty selection state, and begins the voting session by transitioning to state 0 of page 0.

```
4  class Navigator:
▷ 5      def __init__(self, model, audio, video, printer):
6          self.model = model
7          [self.audio, self.video, self.printer] = [audio, video, printer]
8          self.selections = [[] for group in model.groups]
9          self.page_i = None
10         self.goto(0, 0)
```

The **`goto()`** method transitions to a given state and page. It is called by **`invoke()`** and **`timeout()`**. If the transition goes to the last page, the voter's selections are committed. Any state transition (even a transition back to the current state) triggers the playback of the state's audio segments; the **`play()`** method queues the audio instantaneously for later playback. In the ballot definition, `page_i` can be `None` to indicate that no transition should occur; that case is accepted and handled here. Other methods rely on **`goto()`** to always update the video display with a call to **`update()`**, even if no state transition occurs.

```
11     def goto(self, page_i, state_i):
• 12         if page_i != None and self.page_i != len(self.model.pages) - 1:
13             if page_i == len(self.model.pages) - 1:
14                 self.printer.write(self.selections)
15             [self.page_i, self.page] = [page_i, self.model.pages[page_i]]
16             [self.state_i, self.state] = [state_i, self.page.states[state_i]]
17             self.play(self.state.segments)
18         self.update()
```

• *Reviewers found the logic of line 12 confusing, as it combines the "no transition" condition with the "already committed" condition. They all agreed that the navigator should have a flag that indicates whether the votes have already been committed, and a separate method that commits the votes and sets the flag. They also suggested that, to make the commit condition more obvious, the navigator should start on page 1 and always commit on page 0.*

The **update()** method updates the video display based on the current page, state, and selections. It tells the video driver to paste the page's background image over the entire screen, then lay the state's sprite on top of that, and finally fills in any option areas, counter areas, and review areas on the page, in that order. The indices of the slots are assumed to be arranged in sequential order, as described in Chapter 7; hence the variable `slot_i` is incremented in each loop and carried forward to the next loop. Because review areas occupy a variable number of slots depending on their group, the review area loop relies on the **review()** method to return an appropriately incremented value for `slot_i`.

```
19      def update(self):
20          self.video.goto(self.page_i)
21          self.video.paste(self.state.sprite_i, self.state_i)

22          slot_i = len(self.page.states)
23          for area in self.page.option_areas:
24              unselected = area.option_i not in self.selections[area.group_i]
25              group = self.model.groups[area.group_i]
26              option = group.options[area.option_i]
27              self.video.paste(option.sprite_i + unselected, slot_i)
28              slot_i = slot_i + 1

29          for area in self.page.counter_areas:
30              count = len(self.selections[area.group_i])
31              self.video.paste(area.sprite_i + count, slot_i)
32              slot_i = slot_i + 1

33          for area in self.page.review_areas:
34              slot_i = self.review(area.group_i, slot_i, area.cursor_sprite_i)
```

The **review()** method fills in the appropriate sprites for a review area. The arguments `group_i` and `cursor_sprite_i` are parameters of the review area; `slot_i` should be the index of the review area's first slot. The main loop always runs `group.max_sels` times to ensure that `slot_i` cannot go out of range, and that `slot_i` is incremented by the correct amount: `max_sels` × (1 + `max_chars`). Each selected option is pasted into a slot, and then, if the option is a write-in option, a recursive call to **review()** fills in the characters of the write-in. If a cursor sprite is given, it is pasted into the slot just after the last selected option.

```
35      def review(self, group_i, slot_i, cursor_sprite_i):
36          group = self.model.groups[group_i]
37          selections = self.selections[group_i]
38          for i in range(group.max_sels):
39              if i < len(selections):
40                  option = group.options[selections[i]]
41                  self.video.paste(option.sprite_i, slot_i)
42                  if option.writein_group_i != None:
43                      self.review(option.writein_group_i, slot_i + 1, None)
44              if i == len(selections) and cursor_sprite_i != None:
45                  self.video.paste(cursor_sprite_i, slot_i)
46              slot_i = slot_i + 1 + group.max_chars
47          return slot_i
```

- *The reviewers generally found this method to be the most confusing part of the source code, because of its use of recursion and the arithmetic involved in determining `slot_i`. They suggested splitting this into two methods such as **review_contest()** and **review_writein()**; **review_contest()** would call **review_writein()** when necessary. Even though there would be substantial duplication between the two methods, the reviewers felt that eliminating recursion was more important.*

The **press()** and **touch()** methods handle incoming events from the main loop: **press()** handles keypresses and **touch()** handles screen touches. Both methods scan through the bindings of the current state and page, searching for a binding that matches the pressed key or touched target and whose conditions are all satisfied. The first such binding (and only the first such binding) is invoked with a call to the **invoke()** method.

```
▷ 48      def press(self, key):
  49          for binding in self.state.bindings + self.page.bindings:
  50              if key == binding.key and self.test(binding.conditions):
  51                  return self.invoke(binding)

▷ 52      def touch(self, target_i):
  53          for binding in self.state.bindings + self.page.bindings:
  54              if target_i == binding.target_i and self.test(binding.conditions):
  55                  return self.invoke(binding)
```

- *Reviewers felt the method names **press()** and **touch()** were too similar and could be made clearer.*

The **test()** method evaluates a list of conditions and returns 1 only if all the conditions are met. Each of the three predicate types is evaluated in a separate clause; the `cond.invert` flag indicates whether to invert the sense of an individual predicate.

```
  56      def test(self, conditions):
  57          for cond in conditions:
  58              [group_i, option_i] = self.get_option(cond)
  59              if cond.predicate == PR_GROUP_EMPTY:
  60                  result = len(self.selections[group_i]) == 0
  61              if cond.predicate == PR_GROUP_FULL:
  62                  max = self.model.groups[group_i].max_sels
  63                  result = len(self.selections[group_i]) == max
  64              if cond.predicate == PR_OPTION_SELECTED:
  65                  result = option_i in self.selections[group_i]
  66              if cond.invert == result:
  67                  return 0
  68          return 1
```

- *Reviewers felt the comparison of Boolean values on line 66 was "just too clever for its own good." They agreed that lines 66 and 67 could have been more clearly written as*

```
if cond.invert:
    result = not result
if not result:
    return 0
```

to show that `cond.invert` reverses the sense of the condition and that the loop body returns 0 only when the condition is not met.

The **invoke()** method invokes a binding. The steps of the action are carried out, then the audio for the binding is queued, and finally the state transition, if any, takes place. (The **goto()** method handles the case where `next_page_i` is `None`.) Invoking a binding always interrupts any currently playing audio.

```
  69      def invoke(self, binding):
  70          for step in binding.steps:
  71              self.execute(step)
  72          self.audio.stop()
  73          self.play(binding.segments)
  74          self.goto(binding.next_page_i, binding.next_state_i)
```

The **execute()** method executes a single step, which operates on the selection state. It is responsible for ensuring that invalid selection states are never reached.

```
75      def execute(self, step):
76          [group_i, option_i] = self.get_option(step)
77          group = self.model.groups[group_i]
78          selections = self.selections[group_i]
79          selected = option_i in selections

80          if step.op == OP_ADD and not selected or step.op == OP_APPEND:
81              if len(selections) < group.max_sels:
82                  selections.append(option_i)
83          if step.op == OP_REMOVE and selected:
84              selections.remove(option_i)

85          if step.op == OP_POP and len(selections) > 0:
86              selections.pop()
87          if step.op == OP_CLEAR:
88              self.selections[group_i] = []
```

- *Reviewers felt the Boolean expression on line 80 should be clarified with parentheses.*

- *Reviewers found the **execute()** method more confusing than necessary because it uses both the list `self.selections` and a local variable `selections` that aliases a part of it. Mixing these two ways of accessing the list makes it harder to reason about the code, because each could have side-effects on the other. The method would be easier to verify if it always accessed the list through just `self.selections` or just `selections`.*

- *Reviewers felt the method names **invoke()** and **execute()** were too similar and could be made clearer.*

The **timeout()** method handles an inactivity timeout. It is called by the main event loop.

```
▷ 89      def timeout(self):
90          self.play(self.state.timeout_segments)
91          self.goto(self.state.timeout_page_i, self.state.timeout_state_i)
```

The **play()** method plays a list of audio segments. Its job is to translate a list of segments into a sequence of audio clip indices, and send these indices to the audio driver to be queued for playing. Each segment's conditions are checked; if the conditions are met, the corresponding clip index (or indices) are sent to the audio driver. After the clips are queued, **play()** returns immediately; it does not wait for the audio to finish playing, or even to start playing.

```
92      def play(self, segments):
93          for segment in segments:
94              if self.test(segment.conditions):
95                  if segment.type == SG_CLIP:
96                      self.audio.play(segment.clip_i)
97                  else:
98                      [group_i, option_i] = self.get_option(segment)
99                      group = self.model.groups[group_i]
100                     selections = self.selections[group_i]

101                 if segment.type == SG_OPTION:
102                     self.play_option(group.options[option_i], segment.clip_i)
103                 if segment.type == SG_LIST_SELS:
104                     for option_i in selections:
105                         self.play_option(group.options[option_i], segment.clip_i)
106                 if segment.type == SG_COUNT_SELS:
107                     self.audio.play(segment.clip_i + len(selections))
108                 if segment.type == SG_MAX_SELS:
109                     self.audio.play(segment.clip_i + group.max_sels)
```

The **play_option()** method sends audio clips for a given option to the audio driver. There can be multiple clips associated with each option, as dictated by the `option_clips` field of its containing group; the `offset` argument selects which one to play. For a write-in option, this entails playing, in sequence, all the audio clips for the characters in the write-in. Write-in characters are assumed to have only one clip each.

```
110     def play_option(self, option, offset):
111         self.audio.play(option.clip_i + offset)
112         if option.writein_group_i != None:
113             writein_group = self.model.groups[option.writein_group_i]
114             for option_i in self.selections[option.writein_group_i]:
115                 self.audio.play(writein_group.options[option_i].clip_i)
```

The **get_option()** method is used by **test()**, **execute()**, and **play()** to determine the specific group and option for a condition, step, or segment respectively. Conditions, steps, and segments all have fields named `group_i` and `option_i` that can refer to an option either directly or indirectly. When `group_i` is `None`, it's an indirect reference: `option_i` is the index of an option area on the current page. When `group_i` is not `None`, it's a direct reference: `group_i` and `option_i` specify the intended option.

```
116     def get_option(self, object):
117         if object.group_i == None:
118             area = self.page.option_areas[object.option_i]
119             return [area.group_i, area.option_i]
120         return [object.group_i, object.option_i]
```

Audio.py

Audio playback is provided by the `pygame` library.

```
1  import pygame
```

Pygame is based on an event-loop control model. Instead of invoking callbacks, Pygame queues events for processing by the application. Each event has an integer type ID, and Pygame supports user-defined events with type IDs equal to `pygame.USEREVENT` or higher. This module uses `AUDIO_DONE` for signalling when an audio clip has finished playing.

```
2  AUDIO_DONE = pygame.USEREVENT
```

Reviewers suggested that constants like these all be collected in a separate module, and that `main.py` and `Audio.py` refer to the same `AUDIO_DONE` constant instead of redundantly defining it in both files.

The **`Audio`** class is responsible for maintaining a queue of audio clips and causing them to be played in sequence. It ensures that only one clip is playing at a time, and that all the clips are played back one after another until the queue is empty.

```
3  class Audio:
```

The audio driver is initialized with access to the audio section of the ballot definition. It initializes the Pygame audio mixer and converts all the audio clips from raw data into Pygame **Sound** objects. The `playing` flag is exposed to the main program; it indicates whether or not audio is currently playing.

```
▷  4      def __init__(self, audio):
   5          rate = audio.sample_rate
   6          pygame.mixer.init(rate, -16, 0)
   7          self.clips = [make_sound(rate, clip.samples) for clip in audio.clips]
   8          [self.queue, self.playing] = [[], 0]
```

The **`play()`** method puts a single audio clip on the queue. If nothing is currently playing, playback of the given audio clip immediately begins.

```
▷  9      def play(self, clip_i):
  10          self.queue.append(clip_i)
  11          if not self.playing:
  12              self.next()
```

The **`next()`** method takes the next available audio clip off of the queue and starts playing it. The `AUDIO_DONE` event is scheduled to be posted when the audio clip finishes playing. The `playing` member is set to a nonzero value if and only if an audio clip is playing.

```
▷ 13      def next(self):
  14          self.playing = len(self.queue)
  15          if len(self.queue):
  16              self.clips[self.queue.pop(0)].play().set_endevent(AUDIO_DONE)
```

The **`stop()`** method stops audio playback and cancels pending audio.

```
▷ 17      def stop(self):
  18          self.queue = []
  19          pygame.mixer.stop()
```

The **make_sound()** function converts a string of audio data into a Pygame **Sound** object. Because Pygame only knows how to load sounds from files, and the only uncompressed sound format that Pygame accepts is the Microsoft WAVE format, we have to construct a fake file object with a WAVE file header. The header always specifies no compression, monaural audio, and signed 16-bit samples.

```
20 def make_sound(rate, data):
21     [comp_channels, sample_size] = ["\x01\x00\x01\x00", "\x02\x00\x10\x00"]
22     fmt = comp_channels + put_int(rate) + put_int(rate*2) + sample_size
23     file = chunk("RIFF", "WAVE" + chunk("fmt ", fmt) + chunk("data", data))
24     return pygame.mixer.Sound(Buffer(file))
```

The **chunk()** function creates a RIFF chunk, which consists of a 4-byte type code and a 4-byte length followed by a string of data.

```
25 def chunk(type, contents):
26     return type + put_int(len(contents)) + contents
```

The **put_int()** function converts an integer into a 4-byte big-endian representation.

```
27 def put_int(n):
28     [a, b, c, d] = [n/16777216, n/65536, n/256, n]
29     return chr(d % 256) + chr(c % 256) + chr(b % 256) + chr(a % 256)
```

The **Buffer** class is a thin wrapper that makes a string look like a readable file. **make_sound()** wraps this class around the WAVE formatted audio data so it can be passed to Pygame to create a **Sound** object.

```
30 class Buffer:
31     def __init__(self, data):
32         [self.data, self.pos] = [data, 0]

33     def read(self, length):
34         self.pos = self.pos + length
35         return self.data[self.pos - length:self.pos]
```

Video.py

Video display control is provided by the pygame library.

```
1  import pygame
```

The **make_image()** function converts a string containing uncompressed pixel data into a Pygame **Image** object.

```
2  def make_image(im):
3      return pygame.image.fromstring(im.pixels, (im.width, im.height), "RGB")
```

The **Video** class is responsible for pasting full-screen images and sprites onto the display, as well as translating touch locations into target indices.

```
4  class Video:
```

The video driver is initialized with access to the video section of the ballot definition. It initializes the Pygame display and converts all the images from raw data into Pygame **Image** objects. The video driver keeps a pointer to the current layout in its `layout` member so it can look up slots and targets for the current page.

```
▷  5      def __init__(self, video):
   6          size = [video.width, video.height]
   7          self.surface = pygame.display.set_mode(size, pygame.FULLSCREEN)
   8          self.layouts = video.layouts
   9          self.screens = [make_image(layout.screen) for layout in video.layouts]
  10          self.sprites = [make_image(sprite) for sprite in video.sprites]
  11          self.goto(0)
```

The **goto()** method switches to a given layout, which involves pasting the layout's background image over the entire screen.

```
▷ 12      def goto(self, layout_i):
  13          self.layout = self.layouts[layout_i]
  14          self.surface.blit(self.screens[layout_i], [0, 0])
```

The **paste()** method pastes a given sprite into a given slot. The slot coordinates are looked up in the current layout.

```
▷ 15      def paste(self, sprite_i, slot_i):
  16          slot = self.layout.slots[slot_i]
  17          self.surface.blit(self.sprites[sprite_i], [slot.left, slot.top])
```

The **locate()** method finds the target index corresponding to a given touch location. It returns the index of the first enclosing target in the current layout.

```
▷ 18      def locate(self, x, y):
  19          for [i, target] in enumerate(self.layout.targets):
  20              if target.left <= x and x < target.left + target.width:
  21                  if target.top <= y and y < target.top + target.height:
  22                      return i
```

Printer.py

The **Printer** class commits the voter's selections by printing them out. (Other vote-recording mechanisms could be substituted for this module.) It is initialized with access to the text section of the ballot definition.

```
class Printer:
    def __init__(self, text):
        self.text = text
```

The **write()** method does the printing, assuming that the standard output stream is connected to a printer. To prevent any possibility of ambiguous output, the first character of every printed line indicates its purpose, and lines never wrap. An asterisk (*) marks a contest, and a minus sign (-) marks an option. A plus sign (+) marks a write-in group, and an equals sign (=) marks the text of the write-in. A tilde (~) is printed after the name of each write-in character because characters can have names of any length (a feature intended to let ASCII printouts describe write-ins containing non-ASCII characters.) A tilde on a line by itself marks the end of the printout. Here is an example of a printout:

```
* Governor
- Peter Miguel Camejo

* Secretary of State ~ NO SELECTION

* Member of City Council
- William "Bill" G. Glynn
- Write-in 1

+ Member of City Council, Write-in 1
= S~T~E~P~H~E~N~ ~H~A~W~K~I~N~G~

* Proposition 1A
- Yes

~
```

```
    def write(self, selections):
        for [group_i, selection] in enumerate(selections):
            group = self.text.groups[group_i]
            if group.writein:
                if len(selection):
                    print "\n+ " + group.name
                    line = ""
                    for option_i in selection:
                        if len(line) + len(group.options[option_i]) + 1 > 60:
                            print "= " + line
                            line = ""
                        line = line + group.options[option_i] + "~"
                    print "= " + line
            else:
                if len(selection):
                    print "\n* " + group.name
                    for [option_i, option] in enumerate(group.options):
                        if option_i in selection:
                            print "- " + option
                else:
                    print "\n* " + group.name + "   NO SELECTION"
        print "\n~\f"
```