CIS 1100

Beyond `while True`

Python
Fall 2024
University of Pennsylvania

Beyond `while True`

The full rule for `while` loops:

- Test the condition the first time the loop is reached
- If `True`, execute the body of the loop
- If `False`, skip past the body of the loop
- Whenever the last statement of the body of the loop is finished, test the condition again.

In a `while` loop, we can replace `True` with any boolean expression so that we don't loop forever!

Counting Loop

```
count = 0
while count < 10:
  count = count + 2
  print("🐍")
print(count)
```

In box **(S7)**, write the following two numbers:

- How many snakes get printed?
- What's the value of `count` that gets printed?

More *Iteration* to Come

We'll talk more about the rules of loops in the next couple of lectures.

For now, a couple examples of animations that go for a while and then stop.

`rivalry.py`: Animations that Stop

Extending the animation formula a bit, we can think about a recipe for **animations that run until they reach some stopping condition:**

1. Setup: stuff that runs once before the animation/interactivity starts.
2. Animation loop:
    i. Defined with `while <expr>:`, where the expression is something that starts as evaluating to `True` but eventually becomes false
    ii. Clearing & drawing current frame
    iii. Updating variables for next frame
    iv. `pd.advance()`
3. Finale: stuff that runs once **after the animation/interactivity ends.**
    i. Must have `pd.run()` at the end for this to show up.

Example: Guessing Game

Requirements:

- Screen should start blank before any keys are pressed
- Let the game run as long as the player has not pressed the secret key
- Each frame, check if the player has made a guess
  - if the guess is wrong, show the player's guess and tell them they're wrong.
  - if the guess is right, stop the game and display a victory screen.

guessing.py

```
import penndraw as pd
letter = "s"
still_guessing = True
while still_guessing:
  if pd.has_next_key_typed():
    guess = pd.next_key_typed()
    if guess == letter:
      still_guessing = False
    else:
      pd.clear()
      pd.text(0.5, 0.5, f"Not {guess}, try again!")
  pd.advance()

pd.clear(pd.GREEN)
pd.text(0.5, 0.5, f"Hurray! {letter} is right.")
pd.run()
```

Example: Timer

Program should have three "states": before, during, and after press.

- before press:
  - display a red background and nothing else
  - ends when a key is typed for the first time
- during press:
  - starts when a key is typed for the first time
  - continues as long as a key is continuously being held
  - displays the number of consecutive frames that a key has been typed on a green screen
- after press:
  - starts after a key has been released, displays the duration of the press over yellow screen

timer.py

```
import penndraw as pd

button_released = False
button_held = False
counter = 0
pd.clear(pd.RED)
while not button_released:
  if pd.has_next_key_typed():
    button_held = True

  if button_held:
    counter = counter + 1
    pd.clear(pd.GREEN)
    pd.text(0.5, 0.5, f"{counter}...")
    if not pd.has_next_key_typed():
      button_released = True

  pd.advance()

pd.clear(pd.HSS_YELLOW)
pd.text(0.5, 0.5, f"Button held for {counter} frames.")
pd.run()
```

Recap: Sequences

No matter what, all sequence types are ordered collections of elements.

- Ordering gives rise to indexing, which allows for selecting individual elements or subsequences

Different sequence types have different restrictions on what they contain.

- `str`: characters
- `range`: `int` values
- `tuple`: anything
- `list`: anything

Recap: Sequences

Type	Index/Subsequence	Membership	`len()`	Concatenation	Modification
`str`	yes	individual elements or subsequences	yes	yes	no
`range`	yes	individual elements	yes	no	no
`tuple`	yes	individual elements	yes	yes	no
`list`	yes	individual elements	yes	yes	yes (update with `[]`, `append`, `extend`)

Growing Lists: `append`

`append()` allows us to add a single value to the end of a list.

```
numbers_list = [1, 2, 3]

numbers_list.append(4)
print(numbers_list)
```

Prints:

```
[1, 2, 3, 4]
```

Improving `guessing.py`

I want to modify `guessing.py` so that **all previous guesses are saved.** (C12)

```
import penndraw as pd
letter = "s"
still_guessing = True
history = ""          # ! ! !
while still_guessing:
  if pd.has_next_key_typed():
    guess = pd.next_key_typed()
    if guess == letter:
      still_guessing = False
    else:
      pd.clear()
      # TODO: Save the guess!
      pd.text(0.5, 0.5, f"Not {guess}, try again!") # TODO: Change this line to display prev. guesses
  pd.advance()

pd.clear(pd.GREEN)
pd.text(0.5, 0.5, f"Hurray! {letter} is right.") # TODO: Change this line to show no. of guesses taken
pd.run()
```

Toolkit: `len()` and `+` for string concatenation.

Improving `guessing.py`

I want to modify `guessing.py` so that **all previous guesses are saved.** (C14)

```
import penndraw as pd
letter = "s"
still_guessing = True
history = []          # ! ! !
while still_guessing:
  if pd.has_next_key_typed():
    guess = pd.next_key_typed()
    if guess == letter:
      still_guessing = False
    else:
      pd.clear()
      # TODO: Save the guess!
      pd.text(0.5, 0.5, f"Not {guess}, try again!") # TODO: Change this line to display prev. guesses
  pd.advance()

pd.clear(pd.GREEN)
pd.text(0.5, 0.5, f"Hurray! {letter} is right.") # TODO: Change this line to show no. of guesses taken
pd.run()
```

Toolkit: `len()` and something else for adding values to a list.

Recap: Indexing in Sequences

Sequences in Python are **indexable:** we can refer to values at specific positions in the sequence by their positions.

- first value lives at index `0`
- second value lives at index `1`

```
"indexing"
 01234567
```

Notice that "indexing" is a string with eight characters: since we start counting at `0`, the index of the last character is `7`.

Recap: Slicing

We know how to refer to one position in a sequence at a time with a single index.

- How about a group of positions—a **subsequence**?
- If we want to obtain a subsequence of a larger sequence `s` including all characters starting at index `i` and stopping *before* index `j`, then we can do that by writing `s[i:j]`

```
print("earth"[1:4])      # prints "art" 🖌️
print("earth"[0:3])      # prints "ear" 👂
```

This operation is called **slicing.**

Slicing: Starting and Stopping

When slicing, we always *excluding* the element at the end position:

- `"earth[1:4]"` gives `"art"`, which is the subsequence consisting of characters at positions `1`, `2`, and `3` only.
- For a string `s`, `s[i:j]` will always have a length of `j - i` characters.
- To include the last character in a string of length `n`, use a stop index of `n`

Slicing: Shortcuts

```
title = "crossroads"
# all three examples below give exactly the same value
roads_one = title[5:10]
roads_two = title[5:len(title)]
roads_three = title[5:]

print(roads_one)                                  # prints "roads"
print(roads_one == roads_two == roads_three)      # prints True
```

This last version—`title[5:]`—is a useful syntactical shorthand for getting all characters in `title` at & after index `5`.

Slicing: Shortcuts

```
title = "crossroads"
# both examples below give exactly the same value
cross_one = title[0:5]
cross_two = title[:5]

print(cross_one)                # prints "cross"
print(cross_one == cross_two)   # prints True
```

Can similarly omit the first number to take everything from the beginning.

Activity: Slicing and More

- For a phone number written like `"215-898-3500"`, write a slicing expression that gets the **area code**, or the first three digits. **(S7)**
- Some ~~sociopaths~~ well-adjusted people like to pick up a book and read the first and last sentences. If I have a list of words in a novel called `book`, write an expression that creates a list `abridged` that stores the first and last ten words of that list. **(S8)**
  - Remember: list concatenation, negative indexing to count from back
- A file's extension is the portion of its name that is found after the first `.`, e.g. `py` for `hello_world.py` or `txt` for `readme.txt`. Write one or two lines that give you the extension from a string containing a file's name. **(S9)**
  - Remember: `find()`

Recap: Slicing and Stepping

If you only want every `k`th element of a sequence `s`
starting at index `i` and ending at index `j`, you can write

```
s[i:j:k]
```

```
>>> "AaBbCc"[2:5:2]
'BC'
```

- Start at index `2` ("B"), take that character.
- Take `2` steps forward to index `4`.
- Since index `4` is before stop index `5`, take it. ("C")
- Take `2` steps forward to index `6`.
- Since index `6` is not before stop index `5`, stop.

Recap: Reversing

Omit the start and stop values to get a "slice" of the entire string but in reverse.

```
>>> "stop"[::-1]
'pots'
```

A little confusing to parse *why* that works, but a handy tool to keep in mind.

Activity: More Slicing

- Get `"eee"` from `"sequences"` using slicing. **(S10)**
- What's printed? **(M2)**

```
lst = ["global", "array", "of", "chumps," "loafers", "and", "associates"]
my_slice = lst[::2]
print(len(my_slice))
```

  - A: 0, B: 1, C: 2, D: 3

- What's printed? **(M3)**

```
lst = [4, (4, 5, 6), 8, [9, 10, 100]]
my_slice = lst[1:5:2]
print(len(my_slice))
```

  - A: 1, B: 2, C: 6, D: 8

- In the previous question, what's the value of `4 in my_slice`? **(M4)**
  - A: `True`, B: `False`