As you arrive:

1. Start up your computer and plug it in.
2. ***Log into Angel*** and go to CSSE 120.
   Do the ***Attendance Widget*** –
   the PIN is on the board.
3. Go to the ***Course Schedule*** web page.
   Open the ***Slides*** for today if you wish.
4. Checkout today's project: `Session09_FilesAndRobots`

Session 9

Sequences and Objects, applied to Robots

Sequences

- Review

Objects

- Review

Robots

- Design using
  Procedural Decomposition
- Implement using
  Iterative Enhancement

Checkout today's project:

`Session09_FilesAndRobots`

Are you in the Pydev perspective? If not:

`Window ~ Open Perspective ~ Other` then `Pydev`

Messed up views? If so:

`Window ~ Reset Perspective`

Troubles getting today's project? If so:

No SVN repositories view (tab)? If it is not there:

`Window ~ Show View ~ Other`
then `SVN ~ SVN Repositories`

1. ***In your SVN repositories view (tab), expand your repository*** *(the top-level item) if not already expanded.*
   - If no repository, perhaps you are in the wrong Workspace. Get help.
2. ***Right-click on today's project,*** *then select* ***`Checkout`****.*
   Press ***OK*** *as needed.* The project shows up in the
   `Pydev Package Explorer`
   to the right. Expand and browse the modules under `src` as desired.

1. *Sequence* – what is it (in Python)?

- A ***sequence*** is a type of thing in Python that represents an entire *collection* of things.
- More carefully, it represents a
  - finite • *ordered* • *collection* of things
  - indexed by whole numbers
- Examples:
  - A ***list*** `["red", "white", "blue"]`
  - A ***tuple*** `(800, 400)`
  - A ***str*** **(string)** `"Check out Joan Osborne, super musician"`

There are also types for **UNordered collections** *of things – sets and* **Circles**, *for example. More on these in a subsequent session.*

2. Why are Sequences powerful?

- **A sequence lets you refer to an entire collection using a *single name*.**
- You can still get to the items in the collection, by ***indexing***:

```
colors = ["red", "white", "blue"]
colors[0]      has value "red"
colors[1]      has value "white"
colors[2]      has value "blue"
```

Indexing starts at ZERO, not at one.

- And you can *loop* through the items in the collection, like this:

```
for color in colors:
    circle = zg.Circle(...)
    circle.setFill(color)
```

3. Types of Sequences

- There are currently 6 built-in types of Sequences, in two flavors:

Mutable:
- `list`
- `bytearray`

Immutable:
- `str` (a *string*)
- `tuple`
- `range`
- `bytes`

Mutable: *the collection can change after it is created:*
- *Its items can change.*
- *Items can be deleted and added.*

Immutable: *once the collection is created, it can no longer change.*

The following slides explain that different types of Sequences differ in their:
- ***mutability***
- ***type of things they can contain***
- ***notations*** */ how you make them*
- ***operations*** *that you can do to them*

*These are just the **built-in** Sequence types, that is, the ones that you can use without an* `import` *statement. The* `array` *and* `collections` *modules offer additional mutable Sequence types.*

4. How the types of Sequences differ

Type	What objects of this type can contain	Mutable ?	Notation
list	anything	***Yes***	`[a, b, c]`
tuple	anything	No	`(a, b, c)` *OR* `a, b, c` *but:* `()` `(a,)`
string	Unicode characters	No	`'xyz...'` *OR* `"xyz..."`
bytes	Bytes (integers between 0 and 255)	No	Same as string, but with a `b` in front of the string
bytearray	Bytes (integers between 0 and 255)	***Yes***	`bytearray(`*bytes object*`)` `bytes(`*list of ASCII codes*`)`
range	ranges generated by `range`	No	`range(a, b, c)`

Also, different types of Sequences support different operations – more on this in a forthcoming session

5. Looping through sequences

```
def count_big_items_in_sequence(sequence_of_numbers,
                                big_number):
    """ Returns the number of numbers in the given sequence
       that are bigger than or equal to the given 'big' number. """
    count = 0

    for number in sequence_of_numbers:
        if number >= big_number:
            count = count + 1

    return count

def count_big_items_in_sequence_again(sequence_of_numbers,
                                      big_number):
    count = 0

    for k in range(len(sequence_of_numbers)):
        if sequence_of_numbers[k] >= big_number:
            count = count + 1

    return count
```

One way. Pretty.

Another way. Especially useful when you want to refer to *more than one place in the array in* each iteration of the loop.

6. Accumulating sequences

```
def accumulate_list_using_the_plus_operator(n):
    """ Returns a LIST containing n random numbers. """
    numbers = []

    for k in range(n): #@UnusedVariable
        numbers = numbers + [random.randrange(10)]

    return numbers

def accumulate_list_using_append(n):
    """ Returns a LIST containing n random numbers. """
    numbers = []

    for k in range(n): #@UnusedVariable
        numbers.append(random.randrange(10))

    return numbers
```

One way (above). Works for other types of sequences too – just *use the other sequences notation* instead of list notation.

Another way (above). Runs faster than the first way – can you guess why? A similar approach works for strings: accumulate the string into a LIST of characters (or substrings), then do:

```
string_result = ''.join(accumulated_list)
```

Technical notes to discuss at some point

- How to detect bumps in PyCreate
- How a variable's value can be a function, and how you can use that variable to call that function

Rest of today will proceed as follows:

1. Begin with the ***specification*** – what the robot will do
2. Design using ***procedural decomposition*** – what functions should the implementation define and call?
3. Implement using ***iterative enhancement***:
    1. Make the robot do something
    2. Test whether it does it right.
    3. Repeat the previous two steps until the project is complete

Note that you do NOT have to implement the functions in the order that you designed them or the order in which they appear in the source code.

The next 3 slides lead you through these 3 steps.

Specification – what the robot will do

- ***Make a robot wander:***
  - Move forward a random time at a random speed
  - Spin …
  - Forward and spin (simultaneously) …
- ***Asks the user for the parameters:***
  - Maximum time to move (in seconds)
    - Actual time is randomly chosen between 0 and this maximum
  - Maximum speed to move (in cm/sec for forward, degrees/sec for spin)
    - Actual speed is randomly chosen between 1 and this maximum
- After each action, ***if the robot is bumping into something, go backwards a bit***
- Finally:
  - ***Repeats the wander using the SAME PARAMETERS*** for randomness
  - ***Repeats the wander using the SAME PATH*** (as best it can)

Questions about the specification?

Procedural decomposition

- Work in small groups to design a solution to the problem using ***procedural decomposition*** – what functions should the implementation define and call?
  - List the functions you think of.
  - In a few minutes, we will share answers.
  - Then, you can look at the project to see the functions that we suggest you implement and call.

Iterative enhancement

- What is something that you can get the robot to do right away, and then test whether it worked?
- Then, pick another part of the problem and implement that part
  - For example, you might want to get the BUMPING implemented and tested early, since that is new to you
- Continue until the problem is done!
  - Note that you do NOT have to proceed in the order in which the functions were designed or placed into the source code
  - The key is to implement a LITTLE BIT, and then TEST IT before proceeding.

Rest of today: Work on projects as directed by your instructor.