As you arrive:

1. Start up your computer and plug it in
2. ***Log into Angel*** and go to CSSE 120
3. Do the ***Attendance Widget*** – the PIN is on the board
4. Go to the course ***Schedule Page***
   - From your *bookmark*, or from the *Lessons* tab in Angel
5. Open the ***Slides*** for today if you wish

Types

Sequences

- Especially lists

Outline – Help, Types, and Sequences

- Built-in help

- Types
  - What is a ***type***?
  - Examples of types in Python
  - Variables and types
    - The `type` function
  - Numeric types
    - `int`, `float` – differences
  - Convert one type to another

- Sequences
  - What is a ***Sequence***?
  - Why important?
  - Kinds of Sequences, how they differ
    - Especially `list`
  - Operations that any Sequence can do
  - Special operations for Lists

Plus in-class time working on the above concepts, continued as homework.

Built-in Help

- dir()
- dir(<identifier>)
- help(<identifier>)
- To see which functions are built-in:
  - `dir(__builtins__)`
  - `help(__builtins__)`
  - `help(abs)`
- Help on imported functions
- `import math`
- `help(math)`
- `help(math.atan2)`

Q1

Data types

- *Data*
  - Information stored and manipulated on a computer
  - Ultimately stored as bits – 0s and 1s
- But the type of each data item determines:
  - How to interpret the bits
- *Data type*
  - A particular way of interpreting bits
  - Determines the possible values an item can have
  - Determines the operations supported on items
  - Python types include: *int*, *float*, *str*, *list*, *function*, *tuple*

Finding the type of a data item

- Built-in function *type(<expr>)* returns the data type of any value
- Find the types of:
  - 3 3.0 -32 4//5 64.0/5 "Shrubbery" [2, 3] (2, 3)
- Why do we need different numerical types?
  - Operations on int are more efficient and precise
  - Counting requires int
  - floats provide approximate values, used when we need real numbers

Q2-3

Numeric Types - Summary

- int : integer type
  - Exact values
  - Most operations on two ints will yield an int
- float : real number type
  - Approximate values
  - An operation on float and int always yields a float

```
>>> 5//3
1
>>> 5.0/3
1.6666666666666667
>>> 5/2
2.5
>>> 5/2.0
2.5
>>> 5%3
2
>>> 5%2
1
>>> 5.0//2.0
2.0
```

Q4

Practice with types

- Go to SVN Repository view, at bottom of the workbench
  - If it is not there,
    Window→Show View→Other→SVN → SVN Repositories
- Browse SVN Repository view for
  05-TypesAndLists project
- Right-click it, and choose Checkout
  - Accept options as presented
- Expand the 05-TypesAndLists project that appears in Package Explorer (on the left-hand-side)
  - Browse the modules.
  - Do the exercise in the **1-practiceTypes.py** module

Sequences – outline

1. **What is a Sequence (in Python)? Examples.**
2. **Why are Sequences powerful? *Indexing*.**
3. **What kinds of Sequences are there?**
   - ***List*** ***bytearray*** ***str*** (a string) ***tuple*** ***range*** ***bytes***
4. **How do they differ?**
   - *Mutability, what they can contain, notations, operations*
5. ***Operations* that (almost) every Sequence can do:**
   - The *len* function, *accessing* with a subscript, +, *, *slicing*, …
   - Two types of operations: *functions* and *methods*
   - Variables *reference* their value. *Cloning*.
6. **Extra operations that *Lists* can do**
   - Next time: extra operations that *Strings* can do

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.*

Q5

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 = ...
    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.*

Q6

4a. Mutability

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

- Lists are mutable:

```
colors = ["red", "white", "blue"]
colors[1] = "grey"
colors.append("bob")
```

OK

`colors` becomes
`["red", "grey", "blue"]` then
`["red", "grey", "blue". "bob"]`

- Strings and tuples are NOT mutable:

```
building = "Taj Mahal"
building[2] = "g"
pair = (48, 32)
pair[0] = 22
```

NOT OK.
Gives an error message when executed.

- The following have nothing to do with mutability and are perfectly OK:

```
building = "Sistene Chapel"     pair = (0, 0)     colors = []
building = building.replace("Mahal", "Begum")
```

4b. Things that Sequences can contain

This and the following slides explain that different types of Sequences differ in their:

- ***mutability***
- ***type of things they can contain***
- ***notations** / how you make instances*
- ***operations** that you can do to them*

Type	What objects of this type can contain
list	anything
bytearray	bytes, that is, *integers* between 0 and 255
str **(a string)**	Unicode characters (each 16 or 32 bits, depending on an installation option)
tuple	anything
range	ranges generated by `range`
bytes	Bytes (*integers* between 0 and 255)

A ***bit*** is a 0 or 1.

Each ***byte*** is 8 bits and represents an ASCII encoding of one of the 128 pre-Unicode characters.

Unicode allows for far more than the 128 ASCII characters and is the modern standard. See pp. 132-133 or your text.

If you ever need a list-like thing that holds only (say) int's, check out the ***`array`*** *module.*

Q7

4c. Notation and how you can make *instances*

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

Type	Notation, and how you make an instance (options, but not ALL of the options, are shown here)
list	[*blah, blah, ...*] `list(`*sequence*`)` `[`*expression* `for` *variable* `in` *sequence*`]`

str (a string)	`"the charac'ters"` `'the charac"ters'`
	`'''characte\\rs in a \a string with \xF9`
	`stuff th\o274at br\'eaks across lines.'''`

tuple	(*blah, blah, ...*) *blah, blah, ...* But special cases for 0 or 1 elements: ( ) (*blah*,)
range	`range(`*m*`)` `range(`*m*`, `*n*`)` `range(`*m*`, `*n*`, `*i*`)`

4c. Notation and how you can make *instances* (continued)

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

Type	**Notation, and how you make an instance** (options, but not ALL of the options, are shown here)
bytes	*Same as for strings, but put a* **b** *in front, e.g.* `b"the charac'ters"` `b'the charac"ters'` `bytes(`*list of ASCII codes*`)` *For example,* `b'rat'` *is the same as* `bytes([114, 97, 116])`
bytearray	`bytearray(`*bytes object*`)` `bytes(`*list of ASCII codes*`)`

4d. Operations that you can do to Sequences

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

- You can do the following with *any* Sequence
  - Get its ***length***
  - Get the k^{th} element in the Sequence, for any particular k
    - Or get the m^{th} element through the n^{th} element, for any particular m and n
  - ***Concatenate*** and ***Repition***
  - Check for ***membership***
    - that is, whether or not a given item is in the Sequence
  - ***Compare*** two Sequences
    - to see which is "*smaller*" or whether they are "*equal*"
    - And also get the ***smallest*** and ***largest*** elements in the Sequence

Well, almost any Sequence. `Range` objects can't do some of these. But any `list` or `str` or `tuple` or … can do them all.

This next slides discuss each of these in detail.

4d. Operations that you can do to *any* Sequence: ***len*** and ***splicing***

Let **`x`** be a Sequence (so a list or str or whatever), throughout these examples

> *This is called **indexing**. The first index is ZERO, not one. Hence the last index is* **`len(x) - 1`**, *not* **`len(x)`**.

- Get its ***length***

  `len(x)`

- Get the k^{th} element in the Sequence, for any particular k **`x[k]`** **`x[0] ...`**
  - Or get the m^{th} element through the n^{th} element (but not including the n^{th} one), for any particular m and n

    `x[m:n]`

Continued on the next slide

4d. Operations that you can do to *any* Sequence: ***splicing***

- **`list[m:n]`** returns a new list consisting of `[list[m], list[m+1], list[m+2], … list[n-1]]`
- **`list[:n]`** returns a new list consisting of `[list[0], list[1], … list[n-1]]`
- **`list[m:]`** returns a new list consisting of all elements of `list` beginning with `list[m]`.
- **`list[m:n:k]`**, similar to `range(m, n, k)`, returns a new list consisting of **every k^th^ element** of `list`, starting with `list[m]`.

Q8

4d. Operations that you can do to *any* Sequence: ***concatenation*** and ***Repetition***

Let **x** and **y** be two Sequences (so a list or str or whatever), throughout these examples

- Apply *+* and ***, called *concatenation* and *repetition.*
- ***Put examples here.***
- ***Explain that * makes a shallow copy, as does assignment.***

4d. Operations that you can do to *any* Sequence: ***comparisons***

Let **`x`** and **`y`** be two Sequences (so a list or str or whatever), throughout these examples

- Membership
- ***`Compare for <, >, equality`***
- ***`min and max`***

- ***`Explain that we will return to this when we do IF statements`***

List-specific Operations

- <list>**.append (**<expr>**)**
  - Modifies the list by adding the value of the expression to the end of the list
- <list>**.reverse( )**
  - Modifies the list by reversing the order of its elements
- <list>**.sort( )**
  - Modifies the list by sorting the elements into increasing order
- Why don't these operations work with tuples?
- Do the exercises in the **2-practiceLists.py** module.

Not all expressions return values

- >>> **numList = [2, 5, 7, 2, 8, 4, 2, 6]**
- >>> **c = numList.count(2)**
  >>> **c**
  3
- >>> **r = numList.reverse()**
  >>> **numList**
  [6, 2, 4, 8, 2, 7, 5, 2]
- >>> **r**
- >>> **[r]**
  [None]

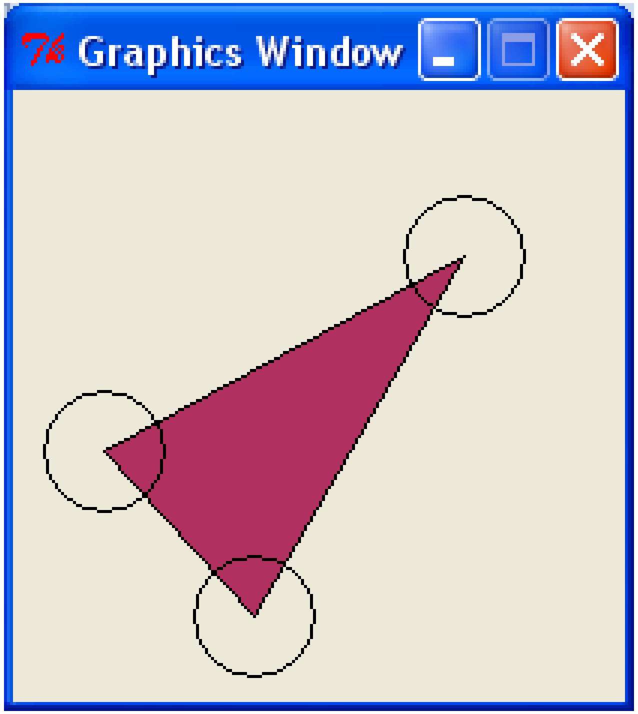
Q9

A List of Points


```
from zellegraphics import *

win = GraphWin()
pointList = [Point(30, 120), Point(150,55), Point(80, 175)]
poly = Polygon(pointList)
poly.setFill('maroon')
poly.draw(win)

for point in pointList:
    circ = Circle(point, 20)
    circ.draw(win)
```



Homework 5

- See instructions linked from Course Schedule
- Upload solutions to dropboxes on ANGEL
- Once you "get the hang" of problems 3 and 4, you should probably start on *Pizza* and *Polygon* while we're here to help

Q10, turn in quiz