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: `Session10_ConditionalsAndFiles`

Session 10

Conditionals and Files

Exam 1 Preview

Conditionals

- Simple and multi-way decisions
- Relational operators
- Boolean operators

Files

- Open, read, write

Checkout today's project:

`Session10_ConditionalsAndFiles`

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 left. Expand and browse the modules under `src` as desired.

Exam 1 information

- **Monday, January 10, 7 p.m. to 9 p.m.**
  - **Olin 267 (Fisher) and Olin 269 (Mutchler)**
- **Format: 2 hours.**
  - Paper part. Resources:
    - Zelle book
    - 1 double-sided sheet of notes that you prepare
  - On-the-computer part. Resources:
    - Zelle book
    - Any written notes that you bring
    - Your computer and the files on it
    - Your own Subversion resources
    - **Any resources you can reach from the course web site by clicking only!**

Possible topics for Exam 1

- Input/compute/output programs
  - Variables, assignment
  - Arithmetic and other expressions
  - input / print,   int / float
- Comments, testing
- Functions:
  - Calling
  - Defining
  - With parameters
  - Returning values
- Definite (for) loops:
  - Through a range
  - Through a sequence
  - Accumulating
    - Summing, Factorial
    - Counting
    - Appending to a sequence
    - Max / Min
- Operations on sequences
  - Lists, Strings, Tuples. Indexing. Slicing.
- Objects
  - Constructing
  - Using methods
  - Accessing instance variables
- Libraries, import
  - math, zellegraphics, time, create
- Decision structures
  - if … elif … else …
  - Relational and Boolean operators
- Files
  - open, read/write, close, parse input

Control structures, Decision structures

- Suppose that you have statements like this:

```
    Blah 1 …
    Blah 2 …
    Blah 3 …
```

  - In what order do they normally execute?
    - Sequentially, one after the other, of course!

Statements that alter the flow are called ***control structures***

- Sometimes we want to alter the sequential flow of a program
  - What examples have we seen of this?
    - ***Loops:*** Repeat execution of a block of code. `for` and `while` statements.
    - ***Function calls and returns:*** Jump to the function. Return to the jump-off point when the function exits.
    - ***Conditionals:***

```
if ...
if ... else ...
if ... elif ... elif ... else
```

Decision structures are control structures that allow programs to "choose" between different sequences of instructions

Next slides discuss these conditional (`if`) statements

Simple decision structure – `if` statement

Decision structures are control structures that allow programs to “choose” between different sequences of instructions

- The `if` statement
  - *Syntax:*
    ```
    if <condition>:
        <body>
    ```
  - *Semantics:* “if the condition evaluates to *true*, execute the body, otherwise skip it”
  - *Example:*

Note the colon

Note the indentation

```
if x < 0:
    print('Illegal input')
    return -1
...
```

What is a “condition”?

What can go between the `if` and the colon? Answer: any expression! But the most typical are generated by:

Next slides discuss each of these.

- ***Comparison operators***

```
if x >= 75:
```

```
if 'dog' in sentence:
```

- ***Functions/methods that return*** **`True`** ***or*** **`False`**

```
if s.islower():
```

Built-in constants.
Note the capitalization.

- ***Boolean operators***

```
if temperature < 32 or temperature > 212:
```

Comparison operators

Note! Why not a single equal sign?

- Traditional

Math	<	≤	=	≥	>	≠
Python	<	<=	==	>=	>	!=

- Set membership
  - `x in y` is true if `x` is a member of the sequence or set `y`
  - `x not in y` is true if `x` is *not* a member of the sequence or set y
- Object equality
  - `x is y` is true if `x` is the same object as `y`

```
a = [1, 2, 3]
b = [1, 2, 3]
a is b
a == b
```

False

True

`==` works like you would think for most objects, but be cautious in using it on floats

Functions/methods that return True/False

```
if s.islower():
    ...
```

Equivalent, but the first form makes more sense when you get used to it

```
if s.islower() == True:
    ...
```

Boolean Constants and Operators

- **Boolean** constants: **True** **False**
- **Boolean** operators: **and** **or** **not**

P	Q	P and Q
T	T	T
T	F	F
F	T	F
F	F	F

P	Q	P or Q
T	T	T
T	F	T
F	T	T
F	F	F

P	not P
T	F
F	T

Truth tables

Example – true or false?

```
not((5 < 9) or not(2 != 7))
```

Do TODO's 1 and 2 in *m1_ifs*

Having It Both Ways: if-else

Syntax:

```
if <Condition>:
      <statementsForTrue>
else:
      <statementForFalse>
```

Semantics:

If <condition> is **True**, execute these statements

If <condition> is **False**, execute these statements

A Mess of Nests

- Can we modify the **grade** function to return letter grades—A, B, C, D, and F?
- Examine `gradeNesting` in *m1_ifs*

Multi-way Decisions

- Syntax:

```
if <condition1>:
    <case 1 statements>
elif <condition2>:
    <case 2 statements>
elif <condition 3>:
    <case 3 statements>
…
else:
    <default statements>
```

reach here if condition1 is false

reach here if condition1 is false AND condition2 is true

reach here if BOTH condition1 AND condition2 are false

Cleaning the Bird Cage

- Advantages of **if-elif-else** vs. nesting
  - Number of cases is clear
  - Each parallel case is at same level in code
  - Less error-prone
- Change **grade** in *m1_ifs* to use **if-elif-else**
- Implement the **gradeFixed** function in *m1_ifs* using **if-elif-else** statement instead of nesting

The *counting* pattern

- A special case of the accumulator pattern
- Example:

```
def count_As(scores):
 """Returns the number of As in the given list of scores"""
    count = 0
    for score in scores:
        if score >= 90:
            count = count + 1
    return count
```

Initialize

Loop

Count conditionally

Files

- Files are *durable* memory – they persist after you shut down your computer (unlike computer RAM).
  - They can be on your hard drive, a USB key, or whatever.
  - The *operating system* is in charge of the *file system*, but programs can ask the operating system to do things with files.
- Key operations on files are:
  - ***Open*** the file
  - ***Read*** from and/or ***write*** to the file
  - ***Close*** the file

Next slides discuss each of these key operations.

We will read/write only ***strings*** from files with ***text***, organized into ***lines,*** processed ***sequentially***, from beginning to end

Google to learn lots more you can do with files:

- Other operations, e.g. deleting a file, listing a folder's contents, or checking if a file exists
- Binary files (instead of text files)
- Random access (instead of sequential access)
- Error handling – e.g., what happens if you try to open a non-existing file for reading

Opening a file

- **Opening** a file makes it available to your program

```
file = open('data.txt', 'r')
```

The `open` function returns a ***stream*** that is used for all subsequent operations on the file.

The name of the file to open – relative to the current folder (as in the example) or absolute, as in `'C:/Program Files/...'` Of course it can be a variable, too.

Either
`'r'` for reading,
or
`'w'` for writing
or
`'a'` for appending
(Other options too.)

Opening a file for reading raises an *Exception* if the file does not exist.

Opening a file for writing *erases* the contents of the file if it already exists!

Closing files

- **Opening** a file makes it available to your program

```
file = open('data.txt', 'r')
```

The *stream* that is used for all subsequent operations on the file.

- **Closing** a stream:

```
file.close()
```

  - Flushes the *buffer* – anything the operating system has not yet written
    - The devices on which files are stored are slow (compared to *main memory*), so changes to the file are often kept in a ***buffer*** in memory and written in clumps (for efficiency) until we close the file or otherwise "flush" the buffer.
  - Tells the operating system that the program is done with the file
    - Causes final "bookkeeping" to happen

Form the habit of closing your files, even though you will often (not always!) get away with not doing so.

Reading from and writing to a file

There are other ways to read files, but this is both efficient and simple.

- One way to *read* from a file is *line by line*:

```
file = open('data.txt', 'r')
for line in file:
    ...
    ...
file.close()
```

The `line` variable here is a *string* whose value is the first line of the file, then the next line of the file, and so forth until the end of the file is reached (and the loop ends).

The `line` variable includes the character(s) that terminate the line.

- One way to *write* to a file is by using the `write` method:

```
file.write(blah)
```

Here `blah` must be a *string*. Each call to `write` appends to the file (i.e., the file is written sequentially).

Exercises on Files

- Do TODO's 1 through 4 in m2_files.py (don't do TODO 5 yet)
  - Make sure that you understand how to:
    - Open a file
    - Read from a file
    - Write to a file
    - Close a file

Parsing the lines that you read

- Our approach to file-reading gives us the data line by line, as a ***string***. But often we want to read ***numbers*** from a file.
  - Extracting parts from a string is called ***parsing*** the string.
- What do we need to extract numbers from a line? Answer: Some way to:
  1. Split the string (line) into words (i.e., strings without spaces)
  2. Interpret a word as an ***int*** or ***float***

The next slides show how to do these string operations.

- Likewise, our `write` method requires a ***string***, so we need a way to convert numbers to strings.

String operations for parsing a line

- Split the string(line) into words (i.e., strings without spaces)

```
s = "This 34 is      a     test.only a &%@!#test!!  "
words = s.split()
```

The above `split` *method sets* `words` *to the* ***list****:*

```
['This', '34', 'is', 'a', 'test.only', 'a', '&%@!#test!!']
```

The `split` method splits on white-space by default, but you can also have it split on other things like commas

- Interpret a word as an ***int*** or ***float***

```
int(r)        float(r)
```

- Convert a number to a string

```
str(x)
```

Do TODO 5 in *m2_files.py*

Rest of Session

- ***Work on today's homework***
  - Ask questions as needed!
- **Sources of help after class:**
  - **Assistants in the CSSE lab**
    - And other times as well (see link on the course home page)
  - **Email**
    - You get faster response from the above than from just your instructor

```
CSSE lab: Moench F-217
7 to 9 p.m.
Sundays thru Thursdays
```

```
csse120-staff@rose-hulman.edu
```