CPSC 231: Getting Started With Python Programming

You will learn basic concepts which apply to all Pythons such as getting user input and displaying output

Python

- This is the name of the programming language that will be used to illustrate different programming concepts this semester:
  - My examples will be written in Python
  - Your assignments will be written in Python
- Some advantages (from Python dot org)
  - Free
  - Powerful
  - Widely used (Google, NASA, Yahoo, Electronic Arts, some Linux operating system scripts etc.)
- Named after a British comedy "Monty Python's Flying Circus"
  - Official website (Python the programming language, not the Monty Python comedy troop): http://www.python.org
  - An overview of the web site: https://www.python.org/about/gettingstarted/

Python History

- Developed in the early 1990's by Guido an Rossum.
- Python was designed with a tradeoff in mind (from "*Python for everyone*" (Horstman and Necaise):
  - Pro: Python programmers could quickly **write** *programs* (and not be burdened with an overly difficult language)
  - Con: Python programs weren't optimized to **run** as efficiently as programs written in some other languages.

"Gawky and proud of it."

From: http://www.python.org/~guido/

Working At Home

- SAFEST APPROACH for working at home (**recommended**).
  - Remotely login to the Computer Science network
  - Example: Connect using a remote login program such as SSH
    - Info: http://pages.cpsc.ucalgary.ca/~tamj/231/starting/ssh_index.html)
    - Download: http://www.ucalgary.ca/it/software/downloads (SSH comes with MacOS so no download is needed.
    - (SSH still needs to be installed but it is easier to install SSH than it is to install and setup Python).
    - There's a learning curve for the interface but it will will be the same in the CPSC lab vs. working at home.
    - *Sometime later in the semester*, in tutorial, the Teaching Assistants will show you how to use this program
    - (Students who took CPSC 101 will have likely learned how to use SSH).
- Alternative (**not recommended**): Getting Python (if you do this make sure you *get version 3.X* and not version *2.X*)
  - http://www.python.org/download/

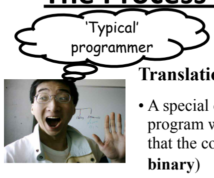
Working At Home (2)

- Alternative (continued):
  - Requires that Python is configured (the "path") on your computer (it is *not mandatory to install Python at home*, follow these instructions carefully, missteps occur at your own peril!)
    - http://docs.python.org/using/windows.html
    - http://docs.python.org/using/unix.html
    - http://docs.python.org/using/mac.html
  - (If you have installed Python on your own computer and still can't get 'Python' to run because of the 'path' problems – this approach works although it's a 'inelegant' solution).
    - Note where you installed Python (folder or directory).
    - Create and run your Python programs from this location.
    - (It's inelegant because you will eventually have a mess from creating many programs all in the same location).

Online Help: Official Python Site

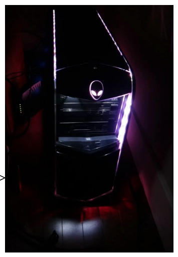
- *Basic explanation* of concepts (for beginners: along with examples to illustrate)
  - http://docs.python.org/py3k/tutorial/index.html
  - (You may want to skip Step #1 and proceed immediately onto Step 2.1 and continue onto Step #3)

The Process Of Creating A Computer Program



Translation

- A special computer program (translator) translates the program written by the programmer into the *only* form that the computer can understand (**machine language/ binary**)



Program Creation

- A person (programmer) writes a computer program (series of instructions).
- The program is written and saved using a text editor.
- The instructions in the programming language (**e.g., Python**) are high level (look much like a human language).

```
# Details later this term
list = [1,2,'a']
for element in list
    print(element)
```

Execution

- The machine/binary language instructions can now be directly executed by the computer.

```
# Details in 2nd year
10000001
10010100 10000100
10000001 01010100
```

(Images curtesy of James Tam)

Types Of Translators

1) Interpreters (e.g., Python is an interpreted language)
   - Each time the program is run the interpreter translates the program (translating a part at a time).
   - If there are any translation errors during the process of interpreting the program, the program will stop execution right where the error is encountered.

2) Compilers (e.g., 'C', C++ are compiled languages)
   - Before the program is run the compiler translates the program all at once.
   - If there are *any translation errors* during the compilation process, no machine language executable will be produced (nothing to execute)
   - If there are *no translation errors* during compilation then a machine language program (e.g., "`.exe`" files) is created which can then be executed.

Location Of My Online Examples

- Finding them via the WWW:
  - URL: http://pages.cpsc.ucalgary.ca/~tamj/231/examples/
- Finding them in UNIX when you are logged onto a computer in the lab (or remotely logged in using SSH)
  - Directory: /home/231/examples
- The locations of the example programs that are ***specific to this section*** of notes (each section will have be located in a sub-directory/sub-link):
  - http://pages.cpsc.ucalgary.ca/~tamj/231/examples/***intro***
  - /home/231/examples/***intro***

–FYI: examples I give TA's for *tutorials* will be in a different location:
  – http://pages.cpsc.ucalgary.ca/~tamj/231/*tutorial*Schedule.html
  – /home/231/*tutorials*

The First Python Program

- Program name: `small.py`

Filename: `small.py`

```
print ("hello",end="")
```

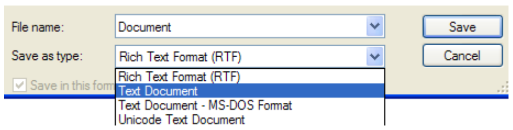
Creating/Running Programs: One Operating System

- The process is similar on other platforms/OS's (the TA's will show you how to do it on the lab computers (Linux) during tutorials).

Step 1: Writing your program

-You need a text editor (e.g., *WordPad*, Notepad) to enter the program.

-It can be done using any editor that you want, but don't use a word processor (e.g., MS-Word) and remember to **save it as a text file** ending with the suffix dot-py "`.py`"



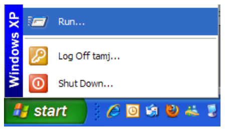
File name: Document
Save as type: Rich Text Format (RTF)
Rich Text Format (RTF)
Text Document
Text Document - MS-DOS Format
Unicode Text Document

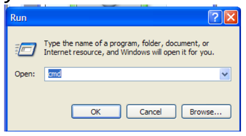
Creating/Running Programs: One Operating System (2)

Step 2: Translating and running your program

-You need to open a command line to translate/run your Python program.

-The name of the Python translator is "`Python`"





Run

Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.

Open: cmd

-To translate/run your program type "`python` *`filename.py`*" at the command line.

- The first example program would be executed by typing "`python small.py`"
- For a program whose filename is called "`output1.py`" you would type "`python output1.py`".

Important Reminders

- Make sure you type the whole file name (including the part after the period) when you translate/run your program.
  - E.g., "`python small.py`"
- Unless you are very familiar with your operating system when you translate/run a program you should first navigate to the directory/folder where your Python program resides.
  - JT: the '`cd`' command changes your directory (Windows and UNIX)
  - Suppose my program was under:

    `C:\231` (Windows)

    OR

    `/home/231` (UNIX)
  - To reach this location you could (shortcuts excluded for now) then *type*:

    `cd` *`c:\231`* (Windows – don't type in the brackets or the stuff in between)

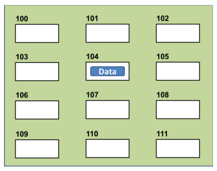
    OR

    `cd /home/231` (UNIX – don't type in the brackets or the stuff in between)

Section Summary: Writing A Small "`Hello World`" Program

- You should know exactly what is required to create/run a simple, executable Python program.
  - While you may not be able to create a new program from scratch at this point, you should be able to enter/run **`small.py`** yourself.

Variables



- Set aside a location in memory.
- Used to store information (temporary).
  - This location can store one 'piece' of information.
    - Putting another piece of information at an existing location overwrites previous information.
  - *At most* the information will be accessible as long as the program runs i.e., it's temporary
- Some types of information which can be stored in variables include: integer (whole), floating point (fractional), strings (essentially any characters you can type and more)

Format (creating):

`<name of variable>` `=` *`<Information to be stored in the variable>`*

Examples (creating):

- Integer (e.g., `num1 = 10`)
- Floating point (e.g., `num2 = 10.0`)
- Strings: alpha, numeric, other characters enclosed in quotes.
  - e.g., `name = "james"`
  - To be safe get in the habit of using double quotes

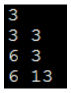
Image curtesy of James Tam

The Assignment Operator: =

- The assignment operator '=' used in writing computer programs does not have the same meaning as mathematics.
  - Don't mix them up!
- Example:

```
y = 3
x = y
x = 6
y = 13
```

- What is the end result? How was this derived (what are the intermediate results)?
- See the program '`assignment.py`'



```
3
3 3
6 3
6 13
```

Variable Naming Conventions

- Python requirements:
  - Rules built into the Python language for writing a program.
  - Somewhat analogous to the grammar of a 'human' language.
  - If the rules are violated then the typical outcome is the program cannot be translated (nor run).
    - A language such as Python may allow for a partial execution (it runs until the error is encountered).
- Style requirements:
  - Approaches for producing a well written program.
  - (The real life analogy is that something written in a human language may follow the grammar but still be poorly written).
  - If style requirements are not followed then the program can still be translated but there may be other problems (more on this during the term).

Variable Naming Conventions (2)

1. Style requirement: The name should be meaningful.
2. Style and Python requirement: Names *must* start with a letter (Python requirement) and *should not* begin with an underscore (style requirement).
3. Style requirement: Names are case sensitive but avoid distinguishing variable names only by case.

Examples

#1:

`age` (yes) `x, y` (no)

#2

`height` (yes) `2x, _height` (no)

#3

`Name, name, nAme` (no to this trio)

Variable Naming Conventions (2)

4. Style requirement: Variable names should generally be all lower case (see next point for the exception).
5. Style requirement: For names composed of multiple words separate each word by capitalizing the first letter of each word (save for the first word) or by using an underscore. (Either approach is acceptable but don't mix and match.)
6. Python requirement: Can't be a keyword (see next slide).

Examples

#4:

`age`, `height`, `weight` (yes)

`Age`, `HEIGHT` (no)

#5

`firstName`, `last_name`

(yes to either approach)

Key Words In Python[1]

and	as	assert	break
class	continue	def	del
elif	else	except	exec
finally	for	from	global
if	import	in	is
lambda	not	or	pass
print	raise	return	try

1 From "*Starting out with Python*" by Tony Gaddis

Variable Naming Conventions: Bottom Line

- Both Python and style requirements should be followed when creating your variables.

Extra Practice

- Traces:
  - Modify the examples (output using format specifiers and escape codes) so that they are still valid Python statements.
    - Alternatively you can try finding some simple ones online or from a textbook.
  - Hand trace the code (execute on paper) without running the program.
  - Then run the program and compare the actual vs. expected result.
- Program writing:
  - Write a program the will right-align text into 3 columns of data.
  - Write a program the will left-align text into 3 columns of data.

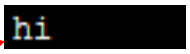
Section Summary: Variables

- What is a variable?
- What are some types of variables available in Python?
- How to create a variable in Python?
- What are naming conventions for variables?

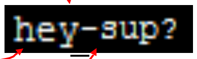
Displaying Output Using The `print()` Function

- This function takes zero or more arguments (inputs)
  - Multiple arguments are separated with commas
  - `print()` will display all the arguments followed by a blank line (move the cursor down a line).
    - end="" isn't mandatory but can be useful to prevent Python from adding the extra line (when precise formatting is needed)
  - Zero arguments just displays a blank line
- Simple Examples (`output1.py`)

```
print("hi")

print("hey",end="")

print("-sup?")
```



```
hi
```



```
hey-sup?
```

Displaying Output Using The `Print()` Function (2)

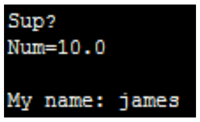
Format:

```
print(arg1,arg2 … )¹
```

Example: `output2.py`

```
num = 10.0
name = "james"
print("Sup?")
print("Num=", end="")
print(num)
print()
print("My name: ", name)
```

Exercise: remove these and see if you can correctly predict the results.



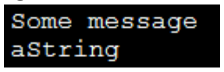
```
Sup?
Num=10.0

My name: james
```

1 From what you've learned thus far each argument can be a constant string or name of a variable.

`Print("… ")` Vs. `Print(<name>)`

- Enclosing the value in parentheses with quotes means the value in between the quotes will be literally displayed onscreen.
- Excluding the quotes will display the contents of a memory location.
- Example: `output3.py`

```
aString = "Some message"
print(aString)
print("aString")
```

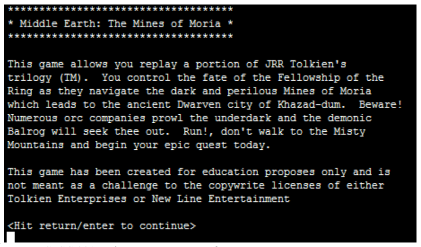


```
Some message
aString
```

Triple Quoted Output

- Used to format text output (free form and to reduce the number of calls to the `print()` function)
- The way in which the text is typed into the program is exactly the way in which the text will appear onscreen.
  - Program name: `formatting1.pyc`

From Python Programming (2nd Edition) by Michael Dawson



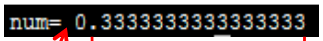
```
************************************
* Middle Earth: The Mines of Moria *
************************************

This game allows you replay a portion of JRR Tolkien's
trilogy (TM).  You control the fate of the Fellowship of the
Ring as they navigate the dark and perilous Mines of Moria
which leads to the ancient Dwarven city of Khazad-dum.  Beware!
Numerous orc companies prowl the underdark and the demonic
Balrog will seek thee out.  Run!, don't walk to the Misty
Mountains and begin your epic quest today.

This game has been created for education proposes only and is
not meant as a challenge to the copywrite licenses of either
Tolkien Enterprises or New Line Entertainment

<Hit return/enter to continue>
```

From a CPSC 231 assignment, courtesy of James Tam

By Default Output Is Unformatted

- Example:

```
num = 1/3
print("num=",num)
```



```
num= 0.3333333333333333
```

$\underbrace{\texttt{0.3333333333333333}}_{\text{The number of places of precision is determined by the language not the programmer}}$

Sometimes you get extra spaces (or blank lines)

The number of places of precision is determined by the language not the programmer

- There may be other issues e.g., you want to display output in columns of fixed width, or right/left aligned output
- There may be times that specific precision is needed in the displaying of floating point values

Formatting Output

- Output can be formatted in Python through the use of format specifiers and escape codes

Format Specifiers

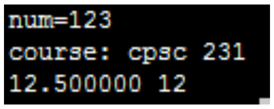
- **Format**:

```
print ("%<type of info to display/code>" %<source of the info
 to display>)
```

Doesn't literally display this: Placeholder (for information to be displayed)

- **Example (starting with simple cases)**:
  - Program name: `formatting2.py`

```
num = 123
st = "cpsc 231"
print("num=%d"        %num)
print("course: %s"    %st)
num = 12.5
print("%f %d" %(num,num))
```



```
num=123
course: cpsc 231
12.500000 12
```

Types Of Information That Can Be Formatted Via Format Specifiers (Placeholder)

Specifier	Type of Information to display
`%s`	String
`%d`	Integer (d = decimal / base 10)
`%f`	Floating point

Formatting Effects Using Format Specifiers

- **Format**:
  `%<width>`[1]`.<precision>`[2]`<type of information>`

- **Examples (format specifiers to format output)**:
  - Program name: `formatting3.p`

```
num = 12.55
print ("%4.1f" %num)
print ("%5.1f" %num)
print ("%3.1f" %num)
print ("%3s%-3s" %("ab", "ab"))
print ("%-3s%3s" %("ab", "ab"))
```

```
12.6
 12.6
12.6
 abab
ab  ab
```

1 A positive integer will add leading spaces (right align), negatives will add trailing spaces (left align). Excluding a value will set the field width to a value large enough to display the output

2 For floating point data only.

One Application Of Format Specifiers

- It can be used to align columns of text.
- Example (movie credits, tabular or financial information)

Section Summary: Formatting Output

- How to use format specifiers (field width, precision) to format output

Escape Codes/Characters

- The back-slash character enclosed within quotes won't be displayed but instead indicates that a formatting (escape) code will follow the slash:

Escape sequence	Description
\a	Alarm: Causes the program to beep.
\n	Newline: Moves the cursor to beginning of the next line.
\t	Tab: Moves the cursor forward one tab stop.
\'	Single quote: Prints a single quote.
\"	Double quote: Prints a double quote.
\\	Backslash: Prints one backslash.

Percent Sign[1]

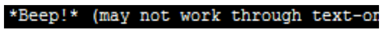
- If no format specifiers are used then simply enclose the '%' within the quotes of a `print()` statement
  `print("12%")` → 12%
- If format specifiers are used within a call to `print()` then use one percent sign to act as an escape code for another percent sign to follow
  `print("%f%%" %(100))` → 100.000000%

1 Since the question inevitably comes up each term I'm answering it now.

Escape Codes (2)

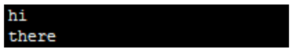
- Program name: `formatting4.py`

print ("\a*Beep!*")



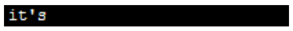
```
*Beep!* (may not work through text-or
```

print ("hi\nthere")



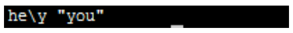
```
hi
there
```

print ('it\'s')



```
it's
```

print ("he\\y \"you\"")

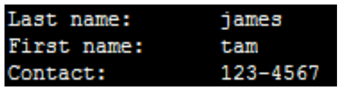


```
he\y "you"
```

Escape Codes: Application

- It can be used to nicely format text output (alignment output, provide separators within and between lines)
- Program example: `formatting5.py`

```
firstName = "James"
lastName = "Tam"
mobile = "123-4567"
print("Last name:\t", lastName)
print("First name:\t", firstName)
print("Contact:\t", mobile)
```



```
Last name:       james
First name:      tam
Contact:         123-4567
```

- Escape codes for aligning text is even more valuable if the width of a field (data to be displayed) is variable e.g., comes from user input or a text file.

Section Summary: Escape Codes

- How to use escape codes to format output

Extra Practice

- Traces:
  - Modify the examples (output using format specifiers and escape codes) so that they are still valid Python statements.
    - Alternatively you can try finding some simple ones online or from a textbook.
  - Hand trace the code (execute on paper) without running the program.
  - Then run the program and compare the actual vs. expected result.
- Program writing:
  - Write a program the will right-align text into 3 columns of data.
  - Write a program the will left-align text into 3 columns of data.

Reminder: Variables

- By convention variable names are all lower case
- The exception is long (multi-word) names
- As the name implies their contents can change as a program runs e.g. (assume '`interest`' and '`bonuses`' are set to valid values)

```
income = 300000
income = income + interest
Income = income + bonuses
```

Named Constants

- They are similar to variables: a memory location that's been given a name.
- Unlike variables their contents *shouldn't* change.
- The naming conventions for choosing variable names generally apply to constants but the name of constants should be all UPPER CASE. (You can separate multiple words with an underscore).
- Example **PI** = 3.14
  - PI = Named constant, 3.14 = Unnamed constant
- They are capitalized so the reader of the program can distinguish them from variables.
  - For some programming languages the translator will enforce the unchanging nature of the constant i.e., giving it a new value is an error
  - For languages such as *Python it is up to the programmer* to recognize a named constant and not to change it.

Why Use Named Constants

1. They make your program easier to read and understand

```
# NO
populationChange = (0.1758 – 0.1257) * currentPopulation
```

Avoid unnamed constants whenever possible!

Vs.

```
#YES
BIRTH_RATE = 17.58
MORTALITY_RATE = 0.1257
currentPopulation = 1000000
populationChange = (BIRTH_RATE - MORTALITY_RATE) *
   currentPopulation
```

Why Use Named Constants (2)

2) Makes the program easier to maintain

- If the constant is referred to several times throughout the program, changing the value of the constant once will change it throughout the program.
- Using named constants is regarded as “good style” when writing a computer program.

Purpose Of Named Constants (3)

```
BIRTH_RATE = 0.998
MORTALITY_RATE = 0.1257
populationChange = 0
currentPopulation = 1000000
populationChange = (BIRTH_RATE - MORTALITY_RATE) * currentPopulation
if (populationChange > 0):
    print("Increase")
    print("Birth rate:", BIRTH_RATE, " Mortality rate:",
 MORTALITY_RATE, " Population change:", populationChange)
elif (populationChange < 0):
    print("Decrease")
    print("Birth rate:", BIRTH_RATE, " Mortality rate:",
 MORTALITY_RATE,  "Population change:", populationChange)
else:
    print("No change")
    print("Birth rate:", BIRTH_RATE, " Mortality rate:",
 MORTALITY_RATE,  "Population change:", populationChange)
```

Purpose Of Named Constants (4)

One change in the initialization of the constant changes every reference to that constant

```
BIRTH_RATE = 0.998
MORTALITY_RATE = 0.1257
populationChange = 0
currentPopulation = 1000000
populationChange = (BIRTH_RATE - MORTALITY_RATE) * currentPopulation
if (populationChange > 0):
    print("Increase")
    print("Birth rate:", BIRTH_RATE, " Mortality rate:",
 MORTALITY_RATE, " Population change:", populationChange)
elif (populationChange < 0):
    print("Decrease")
    print("Birth rate:", BIRTH_RATE, " Mortality rate:",
 MORTALITY_RATE,  "Population change:", populationChange)
else:
    print("No change")
    print("Birth rate:", BIRTH_RATE, " Mortality rate:",
 MORTALITY_RATE,  "Population change:", populationChange)
```

Purpose Of Named Constants (5)

One change in the initialization of the constant changes every reference to that constant

```
BIRTH_RATE = 0.1758
MORTALITY_RATE = 0.0001
populationChange = 0
currentPopulation = 1000000
populationChange = (BIRTH_RATE - MORTALITY_RATE) * currentPopulation
if (populationChange > 0):
    print("Increase")
    print("Birth rate:", BIRTH_RATE, " Mortality rate:",
 MORTALITY_RATE, " Population change:", populationChange)
elif (populationChange < 0):
    print("Decrease")
    print("Birth rate:", BIRTH_RATE, " Mortality rate:",
 MORTALITY_RATE,  "Population change:", populationChange)
else:
    print("No change")
    print("Birth rate:", BIRTH_RATE, " Mortality rate:",
 MORTALITY_RATE,  "Population change:", populationChange)
```

When To Use A Named Constant?

- (Rule of thumb): If you can assign a descriptive, useful, self-explanatory name to a constant then you probably should.
- **Example 1** (easy to provide self explanatory constant name)

```
INCH_CM_RATIO = 2.54
height = height * INCH_CM_RATIO
```

- **Example 2** (providing self explanatory names for the constants is difficult)

```
calories used = (10 x weight) + (6.25 x height) - [(5 x age)
- 161]
```

Extra Practice

- Provide a formula where it would be appropriate to use named constants (should be easy).
- Provide a formula where unnamed constants may be acceptable (may be trickier).
- Search for formulas in science articles online if you are stuck.

Section Summary: Named Constants

- What is a named constant
  - How does it differ from a variable
  - How does it differ from an unnamed constant
  - What are some reasons for using named constants
- Naming conventions for named constants

Arithmetic Operators

Operator	Description	Example	
=	Assignment	num = 7	
+	Addition	num = 2 + 2	
-	Subtraction	num = 6 - 4	
*	Multiplication	num = 5 * 4	
/	Division	num = 9 / 2	4.5
//	Integer division	num = 9 // 2	4
%	Modulo	num = 8 % 3	2
**	Exponent	num = 9 ** 2	81

Order Of Operation

- First level of precedence: top to bottom
- Second level of precedence
  - If there are multiple operations that are on the same level then precedence goes from left to right.

()	Brackets (inner before outer)
**	Exponent
*, /, //, %	Multiplication, division, modulo
+, -	Addition, subtraction
=	Assignment

Example
```
x = 3 * 2 ** 3

Vs.
x = (3 * 2) ** 3
```

Order Of Operation And Style

- Even for languages where there are clear rules of precedence (e.g., Java, Python) it's good style to explicitly parenthesize your operations and use blank spaces as separators.
  ```
  x = (a * b) + (c / d)
  ```
- It not only makes it easier to read complex formulas but also a good habit for languages where precedence is not always clear (e.g., C++, C).

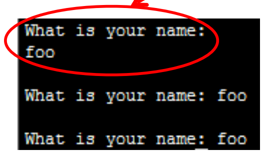
Input

- The computer program getting *string information* from the user.
- Strings cannot be used for calculations (information for getting numeric input will provided shortly).

- **Format:**
  ```
  <variable name> = input()
            OR
  <variable name> = input("<Prompting message>")
  ```

- **Example:** Program name: `input1.py`
  ```
  print("What is your name: ")
  name = input()
        OR
  name = input("What is your name: ")
        OR
  print("What is your name: ", end="")
  name = input()
  ```

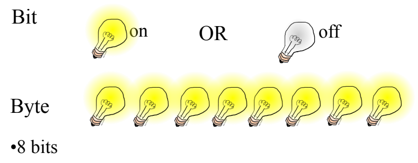
Avoid alignment issues such as this



```
What is your name:
foo

What is your name: foo

What is your name: foo
```

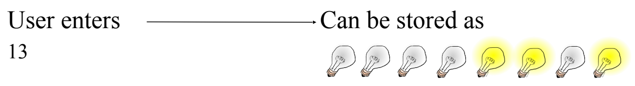
Variables: Storing Information - Optional Details

- On the computer all information is stored in binary (2 states)
  - Example: RAM/memory stores information in a series of on-off combinations
  - A single off/off combination is referred to as a 'bit'



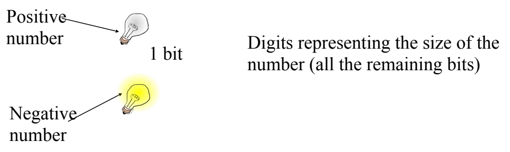
Variables: Storing Information – Optional Details

- Information must be converted into binary to be stored on a computer.



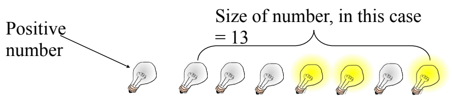
Storing Integer Information – Optional Details

- 1 bit is used to represent the sign, the rest is used to store the size of the number
  - Sign bit: 1/on = negative, 0/off = positive
- **Format:**

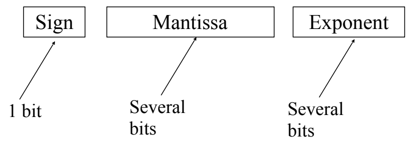


Positive number
1 bit
Digits representing the size of the number (all the remaining bits)
Negative number

- **Previous example**



Storing Real Numbers In The Form Of Floating Point – Optional Details



- Mantissa: digits of the number being stored
- Exponent: the direction (negative = left, positive=right) and the number of places the decimal point must move ('float') when storing the real number as a floating point value.

- Examples with 5 digits used to represent the mantissa:
  - e.g. One: 123.45 is represented as 12345 * 10^{-2}
  - e.g. Two: 0.12 is represented as 12000 * 10^{-5}
  - e.g. Three: 123456 is represented as 12345 * 10^{1}
- Remember: Using floating point numbers may result in a loss of accuracy (the float is an approximation of the real value to be stored).

Storing Character Information – Optional Details

- Typically characters are encoded using ASCII
- Each character is mapped to a numeric value
  - E.g., 'A' = 65, 'B' = 66, 'a' = 97, '2' = 50
- These numeric values are stored in the computer using binary

Character	ASCII numeric code	Binary code
'A'	65	01000001
'B'	66	01000010
'a'	97	01100001
'2'	50	00110010

Storing Information: Bottom Line

- Why is it important to know that different types of information is stored differently?
  - One motivation: sometimes students don't know why it's significant that "123" is not the same as the number 123.
  - Certain operations only apply to certain types of information and can produce errors or unexpected results when applied to other types of information.
- **Example**

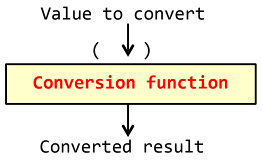
```
num = input("Enter a number")
numHalved = num / 2
```

Converting Between Different Types Of Information

- Example motivation: you may want numerical information to be stored as a string (for built in string functions e.g., check if a string consists only of numbers) but also you want to perform calculations.
- Some of the conversion mechanisms (functions) available in Python:

Format:

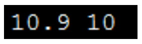
```
int(<value to convert>)
float(<value to convert>)
str(<value to convert>)
```



Examples:

Program name: convert1.py

```
x = 10.9
y = int(x)
print(x,y)
```



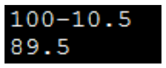
```
10.9 10
```

Converting Between Different Types Of Information (2)

Examples:

Program name: convert2.py

```
x = '100'
y = '-10.5'
print(x + y)
print(int(x) + float(y))
```



```
100-10.5
89.5
```

Converting Types: Extra Practice

- Determine the output of the following program:

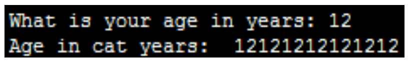
```
print(12+33)
print('12'+'33')
x = 12
y = 21
print(x+y)
print(str(x)+str(y))
```

Converting Between Different Types Of Information: Getting Numeric Input

- The 'input()' function only returns string information so the value returned must be converted to the appropriate type as needed.
  - **Example**
    Program name: convert3.py

```
# No conversion performed: problem!
HUMAN_CAT_AGE_RATIO = 7
age = input("What is your age in years: ")
catAge = age * HUMAN_CAT_AGE_RATIO
print ("Age in cat years: ", catAge)
```

- **'Age' refers to a string not a number.**
- **The '*' is not mathematical multiplication**

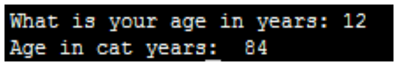


```
What is your age in years: 12
Age in cat years:  12121212121212
```

Converting Between Different Types Of Information: Getting Numeric Input (2)

```
# Input converted: Problem solved!
HUMAN_CAT_AGE_RATIO = 7
age = int(input("What is your age in years: "))
catAge = age * HUMAN_CAT_AGE_RATIO
print("Age in cat years: ", catAge)
```

- **'Age' converted to an integer.**
- **The '*' now multiplies a numeric value.**



```
What is your age in years: 12
Age in cat years:  84
```

Section Summary: Input, Representations

- How to get user input in Python
- How do the different types of variables store/represent information (optional/extra for now)
- How/why to convert between different types

Program Documentation

- Program documentation: Used to provide information about a computer program to another *programmer* (writes or modifies the program).
- This is different from a user manual which is written for people who will *use the program*.
- Documentation is written inside the same file as the computer program (when you see the computer program you can see the documentation).
- The purpose is to help other programmers understand the program: what the different parts of the program do, what are some of it's limitations etc.

Program Documentation (2)

- Doesn't contain instructions for the computer to execute.
- Not translated into machine language.
- Consists of information for the reader of the program:
  - **What does** the program as a whole do e.g., calculate taxes.
  - What are the **specific features** of the program e.g., it calculates personal or small business tax.
  - What are it's **limitations** e.g., it only follows Canadian tax laws and cannot be used in the US. In Canada it doesn't calculate taxes for organizations with yearly gross earnings over $1 billion.
  - What is the **version** of the program
    - If you don't use numbers for the different versions of your program then simply use dates (tie versions with program features – more on this in a moment "Program versioning and backups").

Program Documentation (3)

- **Format:**

```
# <Documentation>
```

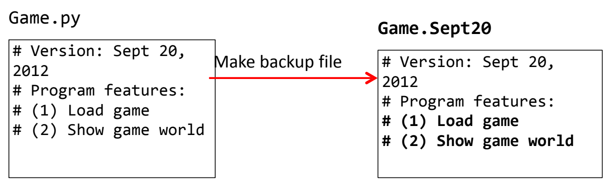
The number sign '#' flags the translator that the remainder of the line is documentation.

- **Examples:**

```
# Tax-It v1.0: This program will electronically calculate
# your tax return. This program will only allow you to complete
# a Canadian tax return.
```

Program Versioning And Back Ups

- As significant program features have been completed (tested and the errors removed/debugged) a new version should be saved in a separate file.



Game.py

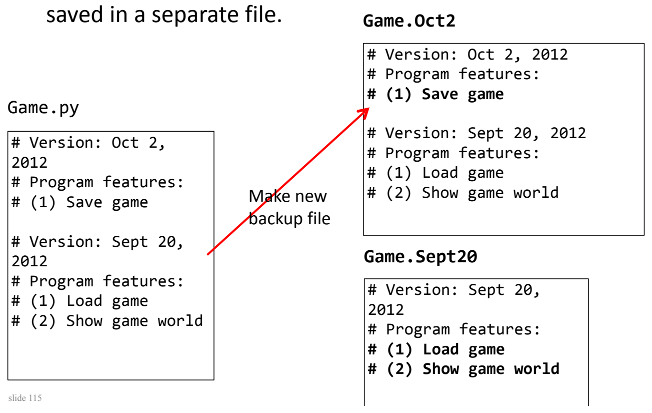
```
# Version: Sept 20,
2012
# Program features:
# (1) Load game
# (2) Show game world
```

Game.Sept20

```
# Version: Sept 20,
2012
# Program features:
# (1) Load game
# (2) Show game world
```

Program Versioning And Back Ups

- As significant program features have been completed (tested and the errors removed/debugged) a new version should be saved in a separate file.



Game.py

```
# Version: Oct 2,
2012
# Program features:
# (1) Save game

# Version: Sept 20,
2012
# Program features:
# (1) Load game
# (2) Show game world
```

Game.Oct2

```
# Version: Oct 2, 2012
# Program features:
# (1) Save game

# Version: Sept 20, 2012
# Program features:
# (1) Load game
# (2) Show game world
```

Game.Sept20

```
# Version: Sept 20,
2012
# Program features:
# (1) Load game
# (2) Show game world
```

Backing Up Your Work

- Do this every time that you have completed a significant milestone in your program.
  - What is 'significant' will vary between people but make sure you do this periodically.
- Ideally the backup file should be stored in a separate directory/ folder (better yet on a separate device and/or using an online method such as an email attachment or 'cloud' storage).
- Common student reason for not making copies: "Backing up files takes time!"
- Compare:
  - Time to copy a file: ~10 seconds (generous in some cases).
  - Time to re-write your program to implement the feature again: 10 minutes (might be overly conservative in some cases).
- **Failing to backup your work is not a sufficient reason for receiving an extension.**

Types Of Documentation

- Header documentation
- Inline documentation

Header Documentation

- Provided at the beginning of the program.
- It describes in a high-level fashion the features of the program as a whole (major features without a great deal of detail).

```
# HEADER DOCUMENTATION
# Word Processor features: print, save, spell check, insert images etc.

<program statement>
<program statement>
```

Inline Documentation

- Provided throughout the program.
- It describes in greater detail the specific features of a part of the program (function, loop, branch, group of related statements).

```
# Documentation: Saving documents
# ‘save’: save document under the current name
# ‘save as’ rename the document to a new name
<program statement>
<program statement>

# Documentation: Spell checking
# The program can spell check documents using the following English variants:
# English (British), English (American), English (Canadian)
<program statement>
<program statement>
```

Over-Documenting A Program

- Except for very small programs documentation should be included
- However it is *possible* to over-document a program
- (Stating the obvious)

```
num = num + 1  # Variable num increased by one
```

- (Documentation can be useful in this case)

```
lastRow = SIZE – 1  # Row numbering begins at zero
```

Section Summary: Documentation

- What is program documentation
- What sort of documentation should be written for your programs
- How program documentation ties into program versioning and backups

Prewritten Python Functions

- Python comes with many functions that are a built in part of the language e.g., ‘`print()`’, ‘`input()`’
- If a program needs to perform a common task e.g., finding the absolute value of a number, then you should first check if the function has already been implemented.
- For a list of all prewritten Python functions.
  - https://docs.python.org/3/library/functions.html
  - Note: some assignments may have specific instructions which list functions you are allowed to use (**assume that you cannot use a function** unless: (1) it’s extremely common e.g., input and output (2) it’s explicitly allowed)
  - Read the requirements specific to each assignment
  - When in doubt don’t use the pre-created code either ask or don’t use it and write the code yourself. (**If you end up using a pre-created function rather than writing the code yourself you could receive no credit**).

Types Of Programming Errors

1. Syntax/translation errors
2. Runtime errors
3. Logic errors

1. Syntax/ Translation Errors

- Each language has rules about how statements are to be structured.
- An English sentence is structured by the *grammar* of the English language:
  - My cat sleeps the sofa.

Grammatically incorrect (FYI: missing the preposition to introduce the prepositional phrase 'the sofa')

- Python statements are structured by the *syntax* of Python:

```
5 = num
```

Syntactically incorrect: the left hand side of an assignment statement cannot be a literal (unnamed) constant (or variable names cannot begin with a number)

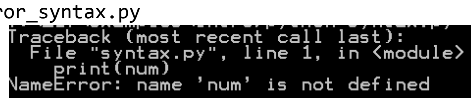
1. Syntax/ Translation Errors (2)

- The translator checks for these errors when a computer program is translated to machine language.

1. Some Common Syntax Errors

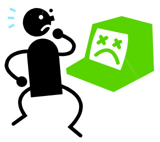
- Miss-spelling names of keywords
  - e.g., '`primt()`' instead of '`print()`'
- Forgetting to match closing quotes or brackets to opening quotes or brackets e.g., **print("hello)**
- Using variables before they've been named (allocated in memory).
- Program name: `error_syntax.py`

```
print(num)
num = 123
```



```
Traceback (most recent call last):
  File "syntax.py", line 1, in <module>
    print(num)
NameError: name 'num' is not defined
```

2. Runtime Errors

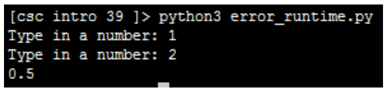


- Occur as a program is executing (running).
- The syntax of the language has *not* been violated (each statement follows the rules/syntax).
- During execution a serious error is encountered that causes the execution (running) of the program to cease.
- With a language like Python where translation occurs just before execution (interpreted) the timing of when runtime errors appear won't seem different from a syntax error.
- But for languages where translation occurs well before execution (compiled) the difference will be quite noticeable.
- A common example of a runtime error is a division by zero error.
  - We will talk about other run time errors later.

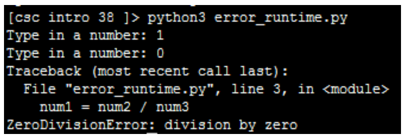
2. Runtime Error[1]: An Example

- Program name: `error_runtime.py`

```
num2 = int(input("Type in a number: "))
num3 = int(input("Type in a number: "))
num1 = num2 / num3 # When zero is entered for num3
print(num1)
```



```
[csc intro 39 ]> python3 error_runtime.py
Type in a number: 1
Type in a number: 2
0.5
```



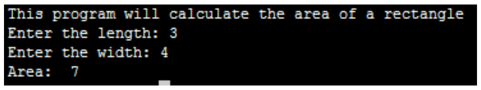
```
[csc intro 38 ]> python3 error_runtime.py
Type in a number: 1
Type in a number: 0
Traceback (most recent call last):
  File "error_runtime.py", line 3, in <module>
    num1 = num2 / num3
ZeroDivisionError: division by zero
```

1 When 'num3' contains zero

3. Logic Errors

- The program has no *syntax errors*.
- The program runs from beginning to end with *no runtime errors*.
- But the logic of the program is incorrect (it doesn't do what it's supposed to and may produce an incorrect result).
- Program name: `error_logic.py`

```
print ("This program will calculate the area of a rectangle")
length = int(input("Enter the length: "))
width = int(input("Enter the width: "))
area = length + width
print("Area: ", area)
```



```
This program will calculate the area of a rectangle
Enter the length: 3
Enter the width: 4
Area:  7
```

Some Additional Examples Of Errors

- All are external links (not produced by your instructor):
  - http://level1wiki.wikidot.com/syntax-error
  - http://www.cs.bu.edu/courses/cs108/guides/debug.html
  - http://cscircles.cemc.uwaterloo.ca/1e-errors/
  - http://www.greenteapress.com/thinkpython/thinkCSpy/html/app01.html

Practice Exercise

- (This one will be an ongoing task).
- As you write you programs, classify the type of errors that you encounter as: syntax/translation, runtime or logical.

Section Summary: The 3 Error Types

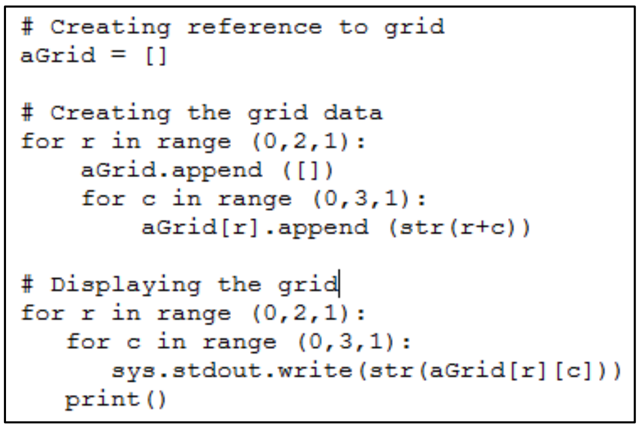
- What are different categories of errors
- What is the difference between the categories of errors and being able to identify examples of each

Layout And Formatting

- Similar to written text: all computer programs (except for the smallest ones) should use white space to group related instructions and to separate different groups.

```
# These are output statements to prompt for user information
Instruction1
Instruction2
Instruction3
Instruction4


# These are instructions to perform calculations on the user
# input and display the results
Instruction5
Instruction6
```

Layout And Formatting: Example



```
# Creating reference to grid
aGrid = []

# Creating the grid data
for r in range (0,2,1):
    aGrid.append ([])
    for c in range (0,3,1):
        aGrid[r].append (str(r+c))

# Displaying the grid
for r in range (0,2,1):
   for c in range (0,3,1):
      sys.stdout.write(str(aGrid[r][c]))
   print()
```

Section Summary: Layout And Formatting

- Why is layout and formatting of programs important, how to do it

Extra: In Case You're Interested

- Different languages may have unique style guides
- Here is the style guide for Python:
  - http://legacy.python.org/dev/peps/pep-0008/

After This Section You Should Now Know

- How to create, translate and run Python programs.
- Variables:
  - What they are used for
  - How to access and change the value of a variable
  - Conventions for naming variables
  - How information is stored differently with different types of variables, converting between types
- Output:
  - How to display messages that are a constant string or the value stored in a memory location (variable or constant) onscreen with `print()`
- How/why to use triple quoted output
- How to format output through:
  - The use of format specifiers
  - Escape codes

After This Section You Should Now Know (2)

- Named constants:
  - What are named constants and how they differ from regular variables
  - What are the benefits of using a named constant vs. unnamed constant
- What are the Python operators for common mathematical operations
- How do the precedence rules/order of operation work in Python
- Input:
  - How to get a program to acquire and store information from the user of the program
- What is program documentation and what are some common things that are included in program documentation
- The existence of prewritten Python functions and how to find descriptions of them

After This Section You Should Now Know (3)

- What are the three programming errors, when do they occur and what is the difference between each one
- How to use formatting to improve the readability of your program

Copyright Notification

- "Unless otherwise indicated, all images in this presentation are used with permission from Microsoft."