CS 115 A First Look at Python

Taken from notes by Dr. Neil Moore

Getting Python and WingIDE

Instructions for installing Python and WingIDE 101 are on the web page:

http://www.cs.uky.edu/~keen/help/installingpython.html

We'll use WingIDE today.

Hint: use a big font (18 or 20 point) for labs! It is easier both for us and for your teammates to read it! The simplest way is **Control + plus** to make bigger.

Structure of a Python program

- `def main():`
  - This is the first line of the "main function" where the program does all its work
    - For now
  - More about functions in a few weeks
  - Python does not *need* a main function, but use on in all code in this class!
    - It's good practice for later.
- Indentation and blocks
  - Code is arranged in indented blocks.
  - The **body** of the main function is one block.
  - It can have several blocks inside it.

Structure of a Python program

- The last line in the file is **main()**
  - This is the **call** to the main function.
  - It is **not** inside the main function!
    - The line (the call) is not indented at all!
  - **If you forget this line, the program does nothing when run!**

Documentation (Comments)

- Syntax: Comments in Python start with a # character and extend to the end of the line.
  - A variant of a comment starts and ends with 3 single quotes. This version can include multiple lines, even paragraphs or pages.
- Semantics: Does nothing: ignored by the Python interpreter entirely.
- Why would we want to ignore any code?
- Comments are for *humans,* not the computer.
  - Your teammates
  - Your boss (or instructor or grader …)
    - You can communicate with your grader while they are grading!
  - Yourself next week! Or next month!

Where to use comments

- Comments don’t usually need to say *how* you are doing something or *what* you are doing.
  - That is what the code is for. Don’t repeat the code in the comments
- Instead, they should say *why something is done*.
  - BAD: counter = 0 # set counter to zero
  - GOOD: counter = 0 # initialize number of lines
- If the comment is long, put it on a line of its own **before** the code statement.
  - That way you don’t have to scroll horizontally to read it all.
    - In general, try to keep code lines less than 80 characters.
    - Less than that on team labs, where you are using a big font.

Where to use comments

- Not every line of code needs its own comment
- A block of code can be summarized by one comment
- Every control structure (loops, if statements) deserves a comment
- Any “tricky” code deserves a comment

Header Comments

- Name, email, section number
- Purpose of the program
- Preconditions: inputs to the program
  - And what the program assumes is true about the inputs
- Postconditions: outputs of the program
  - And what you can guarantee about the outputs
- Reference(s) or Citations when you received or gave assistance
  - TA Name and email
  - Tutor Name and email
  - Partner’s name and email and section
  - URL and date you read the page

Kinds of Errors

Here’s a simple program – it has several errors.

```
Def main():
    x = int(input(“enter a number “))
    x = x + 1 # x should be increased
                  # by 10
    print(x)
Main()
```

- Syntax errors
- Semantic (logic) errors
- Run-time errors

Syntax errors

- Syntax is the set of rules that say how to write statements in the language
  - Misspelling, incorrect punctuation, words in the wrong order, etc. are syntax errors
  - Humans can probably figure out what you meant when you have syntax errors in English (e.g., text messages – misspellings, missing words, no punctuation, etc. but we can still understand them)
  - Programming languages are very rigid about syntax rules – if one exists, the interpreting stops!
  - For computers, getting the meaning if the syntax is wrong is nearly impossible!

Syntax errors

- The interpreter will give you an error message for the first syntax error.
  - Translator programs are NOT “smart” . Their indication of **where** they think the error is is not always right.
  - If they say it’s in line 10, make sure to look in line 9 or 8 or 7 …
  - Don’t bother to look **after** the line they indicate (like line 11 or 12…).
  - If there are comments between lines, skip those and look above them.

Semantic errors

- Also known as **logic errors**
- Semantics = meaning
  - The semantics of a program is what does it make the computer do when it is executed: what changes does it make in memory, what does it output…
- A semantics error is usually the program not doing what you **want** it to do
  - It always does what you **tell** it to!
  - Maybe you multiplied instead of dividing
  - Or you used the wrong variable or constant

Semantic errors

- The interpreter **won’t** detect these for you!
- So how do we find them?
  - Testing!
  - Making a test plan: what to test, provided input, expected output.
  - Coming up with a good set of test cases is one of the important parts of programming
  - By writing up test cases, you have to dig in and understand the desired behavior of the program

Run-time errors

- These occur when the program or interpreter encounters a situation it can’t handle
  - Usually causes the program to halt with an error message, it “crashes”
  - It’s not detected until the situation actually happens!
- Often caused by the environment (operating system):
  - A file is not found
  - Network connection closed
  - A storage device runs out of room
- Sometimes they are caused by programming errors:
  - Using a string where a number was expected
  - Using an undefined variable
  - Dividing by zero
- Some languages allow for catching and handling these errors by using **exception handling** (We’ll do a bit at the end of the semester)

Run-time errors

- For the present time, we will not worry about the errors caused by the environment
- If your program needs a positive number to operate correctly and the user inputs something else, right now it is **alright** for the program to crash
- Your documentation should state the expectations of the program
- As you learn more of the language, you will learn how to catch these errors in friendlier ways

Fixing bugs

- Let’s fix the bugs in our program
  - Syntax error: misspelled keyword
  - Syntax error: name ‘Main’ not defined
  - Semantic error: wrong constant for adding to x
  - Run-time error: input is a string, not a number

Variables

- A variable is a “slot” or “holder” or “location” that refers to a value
  - a and b were variables in our program
  - A value is something like 42 or “Hello”
  - Variables are stored in RAM
  - They can refer to different values as the program runs (they are “able to vary”)
    - **Assignment** (the equals sign) makes a variable refer to a new value
  - A variable is a fundamental building block of most programming languages.

Properties of a variable

- It has a name – one that means something
  - Also called an “identifier”
- It has a value – what value is in the variable
  - In Python, the value of a variable is an **object**.
- It has a type – what **kind** of value
  - Integer, string, floating-point number, boolean, …
- It has a scope – where in the program is the name valid or accessible?
  - In Python, scope goes from the definition of the variable to the end of the block that the definition is in.
  - Can have variables with the same name as long as their **scopes** don’t overlap. They’re entirely unrelated variables!

Rules for Identifiers

- An identifier is a sequence of letters, digits and underscores (_) used as a label
  - “Alphanumeric” characters (“A..Za..z0..9”)
  - **Case sensitive:** students and Students and STUDENTS are all different labels in Python
  - It cannot start with a digit (Python thinks that it is a number, although a badly formatted number)
  - Cannot be a **reserved word** (if, while, else, etc.)
    - These are usually dark blue in WingIDE.

Rules for Identifiers

- Valid examples: x, size, name2, long_name, CamelCase, _ugly (can start with an underscore)
- BAD: 2bad4u, no spaces, no-punctuation!
- Just because it’s legal doesn’t mean it’s a good name.
  - Avoid single-letter variables
    - Except in loop counters or simple math equations
  - And names like “thing” and “number” aren’t any better – they don’t say what they mean
  - Better names are “lineCounter” or “num_students”

The Assignment operator

- Syntax: *variable = expression*
  - Must be a single variable on the left (for now)
- Semantics: Calculates the value of (**evaluates**) the right hand side (RHS) then uses that value to change (replace) the value of the variable on the left hand side (LHS).
- This statement is **not** the same thing as an equation in math!
  - In math, *x = x + 1* has no sensible solution
  - But in Python, x = x + 1 means “add 1 to x”.
  - Instead of “equals”, it’s better to read it as “gets” or …
  - “Assign x + 1 to x” or “Assign x with x + 1”.
- Although it looks trivial, it is where **much** of the processing of the program takes place! It is the most used statement to manipulate items in memory.

The Assignment operator

- Order in the statement matters!
  - The two steps are **always** done in the same order
  - First evaluate the right hand side
  - Then change *only* the variable on the left hand side
  - x + 1 = x # Syntax Error!
- If the LHS variable doesn’t already exist in this scope, it is created.
  - “Initialization”: give a variable its initial value
- Rule of Thumb: a variable has to appear on the left hand side of an assignment **before** it appears on the right hand side (not 100% true but very nearly)

Example of assignment: swapping

Suppose we have two variables and want to swap their values. This means that each variable’s new value is the other variable’s old value.

- The code should look something like this:

```
x = 10
y = 42
# do something
print(x, y)   # should print 42 10
```

- Will this work?

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

- No! it prints out 42 42 We lost the old value of x!

Two Solutions to swapping

- This one works in any language
  - You need a third variable (temp)

```
temp = x
x = y
y = temp
```

- This one works only in Python but it's cute!

```
x, y = y, x
```

It works by making "implicit tuples" on each side and assigning corresponding values to variables on the left hand side.

Can variable properties change?

- The name and scope of a variable never change.
  - If you think it did, it’s actually a different variable
- In a “dynamically typed” language like Python, the value and type of a variable **can change**
  - With assignment statements: (first a float, then a string)

    ```
    score = 0.0
    score = “incomplete”
    ```
- In a “statically typed” language like C++, the type **cannot** change. It is stated at the start of the program and never changes.
- In Python, it’s less confusing to readers and writers if each variable has ONE type. It gets a type when created; you should stick to that type for the life of the variable in the program.
- One common style: include the type in the variable name
  - Like “user_lst” or “name_str” or “hours_int”

Basic Arithmetic

- The **expression** on the right hand side of the assignment operator can be an arithmetic expression.
- Some arithmetic operators in Python are:
  - ** (exponentiation, “raise to the power of”)
  - * (multiply), / (divide)
  - +, - (add and subtract)
- These are listed in order from higher **precedence** to lower **precedence**
- Of course you can use parentheses to make the order you want explicit:

```
total = price * (tax + 100) / 100
```