CS 115 Data Types and Arithmetic; Testing

Taken from notes by Dr. Neil Moore

Statements

A **statement** is the smallest unit of code that can be executed on its own.

- So far we've seen simple statements:
  - Assignment: sum = first + second
  - Function call: print("hi") # doesn't return a useful value
  - Usually simple statements take up one line
- **Compound statements** are bigger.
  - Like: def, for, if, etc.
  - We'll see more of these in the next few weeks.
- Comments are not statements; they aren't executed.

Expressions

An **expression** is a piece of code that has a value. It is even smaller and more fundamental than a statement.

- Something you would use on the right hand side of an assignment operator is an expression.
- Examples:
  - Literals: 2, 4.59, “Python”
  - Variable names: student_name, total_count
  - Arithmetic expressions: 3 * (5 + x)
    - (5 + x) is itself an expression
    - And so are x and 5
    - It’s expressions built of expressions!
  - Function call: input(“What is your name?”) # returns a value
- Expressions are parts of statements, they should not stand alone!

Data Types

Inside the computer, everything is expressed in bits. A **data type** says how to interpret these bits, and what we can do with them. Every expression in Python has a **data type**. Some of the built-in types are:

Type	Description	Examples
int	Integer numbers	2, -44, 0
float	Floating-point numbers	3.0, -0.1, 6.22e23
bool	Boolean (True/False) values	True, False
str	Strings of characters	“hi”, “1234”, “2@5”
list	Lists of values	[“Prisoner”,7], [2, 3, 4, 5, 7]

Integers

The data type **int** represents integers: whole numbers that are positive, zero or negative.

- Literal integers: a sequence of digits, like 2341
  - With no leading zeros!
  - 0 by itself is okay, 007 is not.
- In Python, integers have no stated limit to their size.
  - They can have as many digits as you have memory for.
  - That is not true for most languages, like C++ and Java. They can **overflow** and crash if the numbers get too big!

Floating-point

The data type called **float** represents floating-point numbers, numbers with a decimal point.

- In a computer, they have a wide but limited precision and range.
- Two forms of literal floating-point numbers:
  - A number with a decimal point: 3.14, .027,1., 0.1
    - Must have a decimal point!
    - 1.0 or 1. is a float, 1 is an integer
  - Scientific notation ("E" notation)
    - 6.022e23, 1.0E9, 31e-2
    - The "e" represents "times 10 to the" or "how many places to move decimal"
    - Does not have to have a decimal point if has an E
    - The exponent must be an integer
- In some languages, these are called "doubles".
- Why are they called "floating" point? Water??

Floating-point limitations

- Floats are stored in a binary form of scientific notation:
  - **Mantissa**: the digits (in binary)
  - **Exponent:** how far to move the decimal point
- In Python, the mantissa holds about 15 significant digits.
  - Any digits past that are lost (rounding error).
    - (leading and trailing zeros don't count, they are not significant)
  - This limits the **precision** of a float
  - Try: 10000000000000002.0 – 10000000000000001.0
    - Python's answer is 2.0: the 1 was lost to rounding error!
- The exponent can go from about -300 to 300.
  - Limits the **range** of a float.
  - Try: 1e309
  - It gives inf (infinity)
  - Try: 1e-324
  - It gives 0.0

Floating-point limitations

- The exact limits are on the number of bits, not digits.
  - Even **0.1** can't be represented exactly **in binary**
    - Try: 0.1 + 0.1 + 0.1
    - It gives 0.30000000000000004
- Note that this is NOT the fault of a flaw in the hardware or software or language or OS. It is inherent in trying to store numbers in a **finite** machine. Take CS 321 – Numerical Analysis – one chapter is on studying errors just like this and how to minimize (not eliminate!) them.
- What to take away from all this? Don't expect exact numbers using floating point representation. You won't get it.

Arithmetic on integers and floats

You can perform arithmetic on both ints and floats. For most arithmetic operators (+ - * **) the rules are:

- If both operands are ints, the result is an int.
  - 3 + 5 → 8
  - 2 ** 100 → 1267650600228229401496703205376
- If one operand is a float or both are floats, the result is a float.
  - 3.0 + 0.14 → 3.14
  - 100 – 1.0 → 99.0
  - 2.0 ** 100 → 1267650600228229401496703205376.0
- There is ONE exception…
  - What should 1 / 2 result in?

Division

Python actually has *two* division operators, / and //.

- / *always* gives a **float** no matter what type of operands it has.
  - 1 / 2 → 0.5
  - 6 / 3 → 3.0
  - 3.0 / 0.5 → 6.0
- // does **floor division**: truncates the answer down to a whole number.
  - If both operands are integers, so is the result.
    - 22 // 7 → 3
    - 1 // 2 → 0
  - If either operand is a float, so is the result.
    - But it still has a **whole-number** value.
    - 22 // 7.0 → 3.0
    - 3.1 // 0.5 → 6.0
- With either operator, dividing by zero is a run-time error!
- Note that this is a behavior new to Python in version 3! Version 2 did something different for division!

Remainder (modulo)

The % operator (modulo or mod) finds the remainder of a division.

- Its possible results are between 0 (inclusive) and the right hand side operand (exclusive). Example: for x % 3, the only results are 0, 1, or 2.
  - 6 % 3 → 0 - 7 % 3 → 1
  - 8 % 3 → 2 - 9 % 3 → 0
- Uses for modulo operator:
  - Even/odd: n is even if n % 2 is zero
  - Picking off digits: n % 10 is the last (rightmost) digit of n
  - "Clock arithmetic"
    - Minutes are mod 60: 3:**58** + 15 minutes = 4:**13**
    - Hours are mod 12: 10:00 + 4 hours = **2:00**
- Python can do modulo on floats.
  - 5 % 2.4 → 0.2 (remainder after 2.4 goes into 5 two times, with remainder 0.2)
  - But it is far, far more common with integers.

A common error

- In algebra it is perfectly normal to write things like “2x” or “4ac”. The operator is implied.
  - It’s multiplication!
- In Python this does not work at all. Both of those expressions would be rejected as invalid identifiers, not as multiplied variables.
- You **MUST** put an asterisk * where you mean two things to be multiplied! Even an expression like “2(a + c)” will not work without an operator!

  You must write it as: 2 * (a + c)

The ^ (caret) operator

- A lot of math books will use ^ to mean "raised to the power of" or sometimes, "times 10 to the power of"
- This is NOT the same as the ** operator in Python.
- The ^ operator in Python is a binary XOR operator, working on individual bits of a number. Definitely does NOT do the same thing as **!!
- Thus, if you use an expression like 10^3, you get 9, not 1000!
- But because ^ is a valid operator in Python, you get no kind of warning or error message.
- Be aware! Good test cases will check this by making sure the output answers are correct!

Rounding

One more numeric function, builtin – so you do NOT have to import math library to use it

- round has **either** one or two arguments
  - If it has just ONE argument, it will round the argument to the nearest integer
    - round(5.2) → 5
    - round (7.9) → 8
  - If it has TWO arguments, the second one is the number of decimal places desired. The first argument's value will be rounded to that number of decimals
    - round (math.pi, 2) → 3.14
    - round (2.71818, 0) → 3.0
    - round (12, -1) → 10

Precedence of operators

There are many, many operators in Python! The few which we have seen are listed in priority order, from highest to lowest.

**

*, /, //, %

+, -

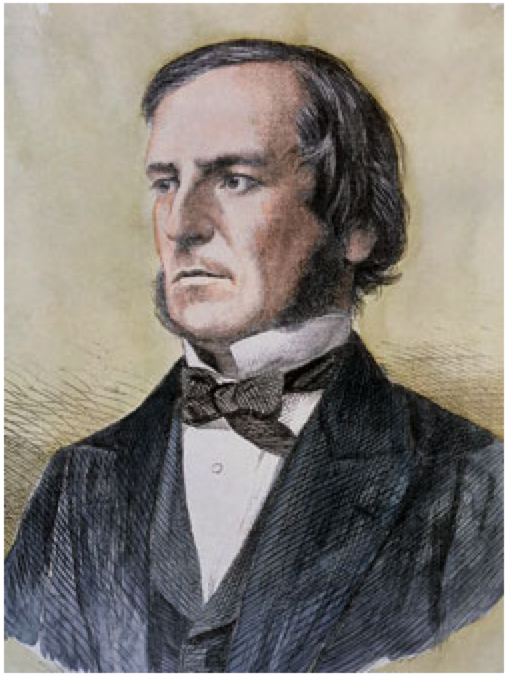
You should learn what each operator does semantically, what types of operands it works ON and what type or types it RETURNS

Booleans

The data type `bool` represents **boolean values.**

- It is named after George Boole, English mathematician and logician. (his picture on next slide)
- Boolean values are the basis of computer circuits: the course EE 280 uses this fact.
- The data type has exactly two values: True and False
  - No quotes! They are not strings.
  - Case sensitive as usual: capital T and F
- You can't do arithmetic with the values
  - The operators you DO use with them are **and, or** and **not.**
  - Most often used with `if` and `while` statements.
  - More on boolean operations in future weeks.

George Boole, inventor of Boolean Algebra (two-valued logic)



Strings

The data type `str` represents **strings**: sequences of characters.

- Literal strings: a sequence of characters in single or double quotes.
  - ‘hello’, “world”, “” (empty string)
  - Use whichever quote isn’t in the string:
    - ‘some “quotes”’, “O’Conner”
- Can perform some operations on strings:
    - **Concatenate** (stick together) strings with a plus (+):
      - `greeting = “Hello, “ + name`
    - Repeat a string by “replicating” with an integer and a *:

```
rating = ‘*’ * 4   # ****
bird = 2 * “do”    # dodo
```

- Can refer to individual elements of strings with subscripts bird[0] is the letter “d”, bird[1] is the letter “o”, bird[2] is the letter “d” again

Escaped Characters

- The **escape** character “\” says to Python, “treat the next symbol specially, not in the normal way”.
- There are some special escaped characters which are useful in strings: tab “\t” and newline “\n”
- If you have to include a single quote character in a string that is delimited by single quotes, **escape** it using a backslash:

```
msg = ‘the word “don\’t” is 5 chars long’
```

- You have to escape backslashes, too:

```
Folder = “C:\\Python 3.4”
```

- All escaped characters are actually ONE character each, even though they are written with two (counting the backslash). Example: “\n\n\n” contains THREE characters.

Converting between types

Converting between data types is also called **type casting.**

- Write the name of the type you are converting to, then, in parentheses, the expression to convert.
  - float(2) → 2.0   int (3.14) → 3 (truncates!)
  - str(1.2e3) → "1200.0"   int("02") → 2
  - float("0") → 0.0   int(" 2 ") → 2 (extra spaces OK)
- Converting float to int rounds towards zero
  - int (-4.2) → -4 and int (4.2) → 4
- You get a run-time error if a string could not be converted:
  - n = int("hello") # CRASHES with ValueError
  - p = int("3.2") # CRASHES, but int(float("3.2")) is OK
- Converting a string **does not** do arithmetic – it does not evaluate first:
  - half = float("1/2") # CRASHES because of the /
  - but half = float("0.5") is OK

Arithmetic and typecasts

- NOTE on arithmetic and typecasts:
  - if you are asked to produce an integer result from a series of steps of calculations, in general, WAIT until you are finished with the calculations before you truncate the result to an integer.
  - Otherwise you are throwing away accuracy!
  - It's the difference between

    int(1.5 + 3.2 + 4.9) = int(9.4) = 9 versus

    int(1.5) + int(3.2) + int(4.9) = 1 + 3 + 4 = 8
  - Of course this may be done in different order if the specification says otherwise.

Output: using print

Every program needs to do output of some kind: to the screen (the Shell window) or a file. In Python, we use the **print** function.

- Sends output to “standard output”.
  - This is usually the shell window, if running inside an IDE
  - Or the command window that appears when you double-click a Python program file (in Windows).
- Syntax: `print(`*`arguments`*`)`
  - `arguments` is a comma-separated list of things to print
    - Can have zero, one or more arguments
  - Each argument can be a literal, a variable, expressions, …
  - Arguments can be any data types: string, integer, float, …
    - `print(“Welcome to my program”)`
    - `print(6 * 7)`
    - `print(“Hello”, name, age)`
    - `print()`

Semantics of print

- Evaluates each argument (computes their values)
- Prints values to standard output, starting at the cursor location
- If multiple arguments are given, a space is put between them
- Outputs a "newline" character after all arguments are printed
  - Moves the cursor to the left end of the next line
  - No-argument `print()` prints just the newline
- The `print` function does not return a value
  - That means you don't use it in an expression:

    ```
    x = print(2)    # BAD, not useful
    ```
  - This is not a syntax error, but x's value will be `None`.
  - Usually this is a semantic error because `None` is a special value, it cannot be used for arithmetic or comparison. It is its own data type.

Extra arguments to print

Sometimes you DON'T want spaces between the arguments when output, or don't want a newline at the end of the output.

- You can control these with so-called **keyword arguments.**
- `sep=`*`string`*`:` Use *string* to separate arguments instead of using a space, the default value.
  - `print(month, day, year, sep='/')`
    - Might output: 1/27/2016
- `end=`*`string`*`:` print *string* at the end, after all the arguments have been printed. This replaces the newline that is printed by default.
  - `print ("The answer is",end=":")`
    `print(answer)` # suppose the variable answer had the value 42
  - would output: `The answer is:42`
  - **This means that the next print statement will start outputting on the same line as where the previous print statement left off.**

Extra arguments to print

- Either string (for sep= or end=) can be empty (nothing between the quotes).
  - `print(first, middle, last, sep="")`
  - output: DLK
- You can use both end= and sep= in the same print statement, but they have to appear at the end of the argument list (in either order).
- If you only have one item to print in that function call, using sep= does nothing. Has to have at least two items to need a separator.

The input function

Most programs also need to get input, usually from the user via the keyboard

- Syntax: `input(prompt)`
  - ONE argument at most (unlike print)
  - The argument is optional: `input()`
- **Returns** (evaluates to) a string (always!)
  - Usually used with the assignment operator

```
name = input(“What is your name? “)
```

Semantics of input

- The `input` function first prints the prompt.
  - Without adding a newline! Usually you should end the prompt in a space, so that the user's input isn't immediately next to the prompt.
    ```
    name = input("What is your name? ")
    ```
  - Include a newline \n in the prompt to get input on the next line: (common style in Zybook)
    ```
    name = input("What is your name?\n")
    ```
  - If no prompt is given, no prompt is printed.
- Pauses the execution of the program, displaying a blinking cursor.
  - Waits for the user to press **Enter**.
- Returns the entire line of input that the user typed, without the newline at the end, as a string.
  - If the user just pressed **Enter** without typing anything, it returns an empty string.

Using the input function

- The function returns a string value.
  - Usually used as the right hand side of an assignment.

    name = input(“What is your name? “)
  - If you don’t put it in an assignment statement, it throws away the input!

    input(“Press Enter to continue”)
  - What if you want numeric input instead of string?
    - Combine it with type casting

      age = int(input(“How many years old are you? “))

      temp = float(input(“What is the temperature? “))
    - What if the input cannot be converted properly to a number?
      - Run-time error (ValueError exception)

Testing programs

We now know enough Python to write a simple program. But how do you know if the program works correctly?

- Testing!
- Verify that the program:
  - Gives the correct outputs
  - Doesn't crash unexpectedly
  - Doesn't run forever (an infinite loop)
- For a four- or five-line program, you could verify it by inspection.
  - But once it gets longer than that, it needs *systematic* testing.
- Some people just plug in some random value and check the output
  - But what if we missed something?
  - We need a PLAN!

Test cases

We will test our programs by trying out a number of **test cases.**

- A typical test case has four parts:
  - Description: what are you testing?
  - Input data you will give to the program
  - The expected output or outcome or behavior from that input
  - The actual output or outcome or behavior from that input

Test cases

- Do the first three parts **before** writing the program
  - Then fill out the actual output by running the program
  - In a software company, the last step is often done by dedicated testers, not the author of the program. (It's hard to be objective about your own code!)
  - In this class, we'll usually omit the last step, "actual output".
    - If it's different from the expected output, you have a bug!
    - And we expect you want to fix the bugs before turning in the program.

Test plan

A **test plan** is a table with a number of test cases.

- Quality is more important than quantity!
- Test cases shouldn't overlap the areas they are testing too much.
  - If all your tests use positive numbers, how will you know whether negative numbers work?
- Making a good test plan requires thought and attention to the problem specifications.
- You should identify and test:
  - Normal cases
  - Special cases
  - Boundary cases
  - Error cases

Sample test plan

Suppose you are writing code to control a vending machine. Inputs are quarters (Q, 25 cents), dollars (D, 100 cents), Coke button (C, costs 75 cents), and Refund button (R).

Description	Inputs				Expected output
Exact change	Q	Q	Q	C	Vend one Coke.
Inexact change	D	C	-	-	Vend one Code, return one quarter
Not enough money	Q	Q	C	-	Flash “need 25 cents”
Enough money, eventually	Q	C	D	C	Flash “need 50 cents”, vend one Code, return 2 quarters
Giving a refund	Q	Q	R	-	Return two quarters.
Refund with no money inserted	R	-	-	-	Do nothing

Off-by-one errors

You need to build a fence 100 feet long, with a fence post every 10 feet. How many posts do you need?

- You need 11, not 10!
- This is a very common source of errors in programming.
  - “Fencepost errors” or “off-by-one errors”

Off-by-one errors

- Whenever your program involves ranges (1-10, letters "L" – "R")
  - Test the **boundary cases**
  - Not just the exact endpoints, but adjacent values
    - So for the first range 1-10, test 0, 1, 2 (lower) and 9, 10, 11 (upper)
    - For the range "L"-"R",   "K","L","M"   and   "Q", "R", "S"
  - Why test boundary cases?
    - It's easy to stop before an endpoint
    - Or to go too far, past the endpoint
    - Make sure in-range inputs are accepted
    - Make sure out-of-range inputs are rejected
    - Make sure the exact boundaries are treated according to the specifications

Regression testing

What happens when you find a bug?

- You're running your tests and you find an error on test #5.
  - So you fix the bug in your program.
  - Now what?
    - Run test #5 again – make sure you actually fixed it!
- What about tests #1 - #4?
  - Those tests passed already, right?
  - But what if your fix broke something?

Regression

- **Regression** is “returning to an earlier, usually lower or less desirable state”
  - Like something that used to work but doesn’t any more.
    - Because you changed something
    - How to avoid regressions?
- **Regression testing:** whenever you change the code, go back to the beginning of the test plan and **repeat ALL the tests in the test plan.**
  - To make sure you didn’t *add* or *uncover* another bug!
  - This will save you many points on CS 115 programs!