More About Values

Casts

- To ***cast*** is to take a value of one type and return the corresponding value of some other type (or an error, if the cast is impossible)
  - **int(*x*)** casts a string, float, or boolean ***x*** to an integer
  - **float(*x*)** casts a string, integer, or boolean ***x*** to a float
  - **str(*x*)** returns the value of ***x*** as a string
  - **hex(*int*)** returns a **string** representing the hexadecimal value of the integer ***int***
  - **oct(*int*)** returns a **string** representing the octal value of the integer ***int***
  - **bin(*int*)** returns a **string** representing the binary value of the integer ***int***
  - **chr(*int*)** returns the character represented by the Unicode value ***int***
  - **ord(*ch*)** returns the integer value of the Unicode character ***ch***
  - unichr(x) is in the textbook, but no longer exists, because in Python 3, all strings are in Unicode

ASCII and Unicode

- On consumer laptops, memory is organized into ***bytes***
  - A ***byte*** is eight ***bits***
  - A ***bit*** is a single on/off (or 0/1, or true/false) value
- For many years, ***ASCII*** (***American Standard Code for Information Exchange***) has been the world standard
  - ASCII uses ***seven*** bits to represent each character
  - This is enough to represent all the characters on a standard (American) keyboard, with one bit left over
  - By also using the leftover ***eighth*** bit, every company could add more characters (like “smart quotes”)--and every company did
    - This is why smart quotes (and em-dashes, and many other characters) typed in Windows turn into other, weird characters on a Macintosh...and vice versa
- ***Unicode*** is a much newer standard (~1988) that solves this problem
  - Unicode uses single bytes to represent ASCII characters, and multiple bytes for any additional characters

Numbers

- In mathematics, ***integers*** (whole numbers) are **exact** and may be **arbitrarily large**
- In programming, integers are **exact** and in Python (but not most languages) they may be **arbitrarily large**

- In mathematics, ***real numbers*** have **infinite precision**
  - π has been calculated to 2.7 trillion digits
- In programming, ***floating point*** numbers have limited precision
  - **>>> 10 / 3**
    3.3333333333333335

Equality

- **Rule:** Never compare floating point numbers for exact equality (**==**) or exact inequality (**!=**)
  - This comparison usually works, but you can't trust it
  - It's safe to compare integers
    - Except:
      ```
      >>> i = 12345678901234567890
      >>> i == int(float(i))
      False
      ```
- **Rule:** If exact results are required, use only integers.
  - It's probably to use floats to keep track of your own dollars and cents
  - Don't do this for banks or other financial institutions!

Approximate equality

- What if you want to determine whether two floating point numbers are approximately equal?
  - You can test if the **absolute value of their difference** is small
    - **abs(x - y) < epsilon**
    - What is **epsilon**?
      - For "ordinary" numbers, maybe **0.00001** is a reasonable value
      - For distances between **atoms**, maybe **1.0e-9** is a better value
      - For distances between **stars**, maybe **1.0e9** is a better value
  - You can test if the **absolute value of their quotient** is near **1**
    - **abs(x / y - 1.0) < epsilon**
    - With this approach, **epsilon** is more like a "percentage" difference, and doesn't have to be adjusted for the expected size of x and y
    - Unfortunately, if **y** is zero, this will cause your program to crash

Short-circuit logic

x	y	x and y	x or y
TRUE	**TRUE**	TRUE	TRUE
TRUE	**FALSE**	FALSE	TRUE
FALSE	**TRUE**	FALSE	TRUE
FALSE	**FALSE**	FALSE	FALSE

- Consider **x and y**: If **x** is **False**, Python doesn’t have to evaluate **y**
- Consider **x or y**: If **x** is **True**, Python doesn’t have to evaluate **y**
- Consider:
  (y == 0) or (abs(x / y - 1.0)) < 0.00001
  - What will this do if **y == 0**?
  - Do you think this is a good way to determine if **x** and **y** are approximately equal?

The ***ternary operator***

- Python has an **if-else** ***operator*** as well as an **if-else** statement
- **Syntax:** *true_expr* **if** *condition* **else** *false_expr*
- **Semantics:** Evaluate the *condition*; if **True**, evaluate the *true_expr*, otherwise evaluate the *false_expr*. Whichever gets evaluated becomes the value of the expression
- **Example:** `abs_x = x if x >=0 else -x`
- **Example:**

```
if (x == 0 if y == 0 else abs(x / y - 1.0) < eps):
    # do something when x and y are almost equal
```

  - The outer parentheses are not necessary in the above, but make it somewhat less difficult to read
  - I think the ternary operator is usually confusing and should be avoided
  - What is the result of this expression if **x == 1.3e-50** and **y == 0**?
  - What is the result of this expression if **x == 0** and **y == 1.3e-50**?

Number systems

- The ***binary*** (base 2) number system uses two "binary digits, " (abbreviation: bits) -- **0** and **1**
- The ***octal*** (base 8) number system uses eight digits: **0, 1, 2, 3, 4, 5, 6, 7**
- The ***decimal*** (base 10) number system uses ten digits: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**
- The ***hexadecimal***, or "***hex***" (base 16) number system uses sixteen digits: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F**
- In Python, write binary, octal, or hexadecimal numbers by prefixing them with **0b**, **0o**, or **0x**, respectively
- Regardless of how they are written, numbers are **stored** in binary and **displayed** in decimal

Using octal and hex

- Computers use binary, but the numbers are too long and confusing for people--it's easy to lose your place
- Octal or hex is better for people
- Translation between **binary** and **octal or hex** is easy
- One octal digit equals three binary digits:
  `101101011100101000001011`
  ` 5  5  3  4  5  0  1  3`
- One hexadecimal digit equals four binary digits:
  `101101011100101000001011`
  `  B    5    C    A    0    B`

Bitwise operators

x	y	~x and y	x & y	x | y	x ^ y
0	**0**	1	0	0	0
0	**1**	1	0	1	1
1	**0**	0	0	1	1
1	**1**	0	1	1	0

- **~** is "bitwise not" (or "invert" or "toggle")
- **&** is "bitwise and"
- **|** is "bitwise or"
- **^** is "bitwise exclusive or" (or "xor")
- **x >> i** shifts the bits in **x** to the right **i** places
- **x << i** shifts the bits in **x** to the left **i** places

Precedence

- **2 + 3 * 4** is **14**, not **20**
- This is because multiplication has higher precedence than addition
- Here's what you should remember about precedence:
  - Exponentiation (******) has highest precedence
  - Unary operators have higher precedence than the related binary operators in the same family
    - By "family" I mean arithmetic, logic, or bitwise operators, so **not x or y** means the same as **(not x) or y**
  - Multiplication, division, and "and" operators have higher precedence than additions, subtraction, and "or" operators
    - **x or y and z** means the same as **x or (y and z)**
- For everything else, use parentheses to clarify your meaning, even if they aren't needed

Comparison chaining

- In Python, you can ***chain*** comparisons
  - Example: `x < y < z` is legal and meaningful
- In every other language that I know, `x < y` would result in a ***boolean*** value, which would then be compared with z
- Comparison chaining is a nice feature, and you should feel free to use it, but...
  - It’s not allowed in Java
  - In C or C++ or C# it is allowed, but `x < y` will result in `0` or `1`, which is then compared with `z`

Shorthand assignments

- **x += y** is shorthand for **x = x + y**
  - Example: **x += 1** adds **1** to **x**
- This same shorthand works for all the other binary operators

The End

- Why do programmers confuse Halloween with Christmas?

- Because Oct 31 == Dec 25.