Variables and Values

Names

- Variables (which hold values) and functions (which are blocks of code) both have **names**
  - Names *must* begin with a letter and *may* contain letters, digits, and underscores
  - Names are ***case-sensitive***—**`total`**, **`TOTAL`**, and **`Total`** are three different names
  - There are a number of reserved words, such as **`if`** and **`while`**, that cannot be used as the name of a variable or function

Style of names

- ***Style*** is all the little things, not required by the language, that make a program easier or harder to read
  - For example: Lines of code should not extend past about column 80, because long lines are harder to read (especially if you have to scroll sideways!)
- Some style rules are just commonly accepted conventions, as in, “This is the way we do things”
- **Style rule:** Variable names should always begin with a lowercase letter
- In Java, names composed of multiple words are (almost) always written in “***camelCase***,” for example, **`sumOfAngles`**
- In Python, camel case is sometimes seen, but much more often, multiword name use underscores, for example, **`sum_of_angles`**
- **Style rule:** In this course, use underscores for Python, camelCase for Java

Importance of style

- Programs are *read* more often than they are *written*
  - Estimates range from 20 times to 50 times
- The easier a program is to read, the easier it is to:
  - Understand
  - Debug
  - Enhance
  - Modify/update
- Style is less important for very small programs (say, less than 1000 lines)
- Without good style, even moderately sized programs become difficult or impossible to debug, let alone enhance or update
- This course is not about writing very small programs!

Strings

- Strings are composed of ***zero or more*** characters
- Like everything else on the computer, characters are represented in ***binary*** (a sequence of zeros and ones)
- Until recently, ***ASCII*** (American Standard Code for Information Interchange) was the most commonly used encoding
  - ASCII allowed for 127 characters; for example, the letter **`a`** was represented by **`01100001`**
  - ASCII was fine for representing English text, digits, and a handful of punctuation marks
- ***Unicode*** is an extension of ASCII that allows for hundreds of thousands of characters
- Python 2 uses ASCII; Python 3 uses Unicode

Writing strings

- Strings may be enclosed in;
  - Single quotes, `'Like this'`
  - Double quotes, `"Like this"`
  - So-called “triple quotes,” `'''Like this'''` or `"""Like this"""`
- You can put double quotes inside a single-quoted string, or single quotes inside a double-quoted string, or either inside a triple-quoted string
- You can put a single quote inside a single-quoted string if you ***escape*** it, like this: `\'`
- The same goes for double quotes inside doubly-quoted strings: `\"`
  - Example: `"She said, \"Don't\""`
- Triply-quoted strings can extend across several lines; other kinds cannot

Additional escaped characters

- Some single characters cannot easily be entered directly into strings, and must be “escaped” (backslashed)
  - `\n` represents a newline character
  - `\t` represents a tab character
  - `\'` represents a single quote (inside a singly-quoted string)
  - `\"` represents a double quote (inside a doubly-quoted string)
- The above do not work inside triply-quoted strings
- Characters not in ASCII, but just in Unicode, are written as **`\u`*hhhh***, where the *h*s are hexadecimal digits (`0 1 2 3 4 5 6 7 8 9 A B C D E F`)
  - Example: `\u03C0` is *π*
- Unicode characters do work in triply-quoted strings
- You can look up the character codes on the web

Ways to write integers

- Integers can be written in ***binary*** (base 2), ***octal*** (base 8), ***decimal*** (base 10) or ***hexadecimal*** (base 16)
- By default, integers are decimal
    - Binary integers are written with an initial **`0b`**
    - Octal integers are written with an initial **`0o`**
    - Hexadecimal integers are written with an initial **`0x`**
        - In a string, Unicode characters are written as **`\u`** followed by four hexadecimal digits
    - Decimal numbers other than **`0`** may not be written with an initial **`0`**

Ways to write floats

- There is seldom any reason to write floating-point numbers in a base other than decimal
- Any number with a decimal point is a floating-point number
  - Examples: **`12.5`**, **`12.`**, **`.5`**
- Any number in scientific notation is a floating-point number
  - Avogadro's number in scientific notation is 6.022×10^{23}
  - Since ASCII had neither the × symbol nor superscripts, we use **`E`** or **`e`** to indicate "...times 10 to the..."
  - Hence Avogadro's number has to be written as **`6.022E23`**

Arithmetic expressions

- Just as in algebra, operations have precedence
  - The unary operators `+` and `–` are done first
  - Next comes exponentiation, `**`
  - Next multiplication (`*`) and division, (`/`, `//`, `%`)
  - Finally addition `+` and subtraction -
- *Parentheses*, `( )`, can be used to alter the order of operations
  - *Brackets*, `[ ]`, and *braces*, `{ }`, **cannot** be used for this purpose
  - If you learned a variant of English where, for example, `( )` were called "brackets," that is ***not*** how these terms are used in programming!

Style in expressions

- Good style:

```
x = -b + sqrt(b ** 2 - 4 * a * c)
```

- Poor style:

```
x=-b+sqrt(b**2-4*a*c)
```

- Just as in English, it’shardertoreadan expression whenthere aren’tspaceswheretheybelong
- **Rule:** Put spaces around all *binary* operators
  - There is no space after a *unary* operator, such as `-b` in the above example, or between a function name and the opening parenthesis
- **Rule:** *Do not* put spaces immediately inside parentheses
  - Your textbook puts spaces here, as for example

    ```
    print( "hello" )
    ```

    but this is ***very*** unusual, and I strongly discourage doing so

Boolean expressions

- Boolean expressions use the literal values **`True`** and **`False`**, and the logical operators **`and`**, **`or`**, and **`not`**
  - **`not`**, being unary, has the highest precedence
  - **`and`** has higher precedence than **`or`**
  - Example: **`p and q or not r`** means the same as **`(p and q) or (not r)`**
- Other operators all have higher priority, so **`not p == q`** means **`not (p == q)`**
  - When in doubt, use parentheses!

Boolean style 1

- In Python, as in some other languages, tests don’t always have to be Booleans
  - Zero and a few other things typically mean “false,” things not considered false mean “true”
    - Example:

      ```
      if a - b:
          print("unequal")
      else:
          print("equal")
      ```

      will print “equal” if a == b
    - This sort of thing is necessary in the C language, which doesn’t have Booleans, but is unnecessary and undesirable in Python, which does have Booleans
    - `if a != b` is much clearer than `if a - b`
      (Remember, `!=` means “not equal to”)
- **Rule:** Only use Booleans for test conditions.

Boolean style 2

- **Rule:** Avoid double negatives.
  - In an `if` statement, this means putting the positive case first
  - Example: Don’t do this:

    ```
    if a != b:
        # What to do when a and b are not equal
    else:
        # What to do when a and b are not not equal
    ```
  - Possible exception: If the negative case is short and the positive case is very long, it may be better to put the shorter case first
- **Rule:** Never compare a Boolean result to `True` or `False`
  - For example, suppose you have a function `isPrime(n)` to test whether a number *n* is prime or not prime (the function returns `True` or `False`). Then
    - You can say `if isPrime(n):`
    - You *could* say `if isPrime(n) == True:` , but it’s redundant and just looks silly

Bitwise operators

- It is sometimes convenient to work with a sequence of ***bits*** (`0`s and `1`s)
- Here are examples of each of the bit operators:
  - Not: `~0b1100 == 0b0011`
  - And: `0b1100 & 0b1010 == 0b1000`
  - Or: `0b1100 | 0b1010 == 0b1110`
  - Exclusive or: `0b1100 ^ 0b1010 == 0b0110`
  - Left shift: `0b00010011 << 2 == 0b01001100`
  - Right shift: `0b01001100 >> 2 == 0b00010011`

Assignment abbreviations

- `=` means assignment: The variable on the left gets the value of the expression on the right
  - Remember, use `==` to test if two things are equal
- `largestValue = largestValue + increment`
  may be abbreviated to
  `largestValue += increment`
- `largestValue = largestValue - increment`
  may be abbreviated to
  `largestValue -= increment`
- ...and similarly for all the other operators
- `bitSequence = bitSequence & mask`
  may be abbreviated to
  `bitSequence &= mask`
- Etc.

The End

- Give a person a program, and you frustrate them for a day;

  Teach a person to program, and you frustrate them for a lifetime.

  --Anonymous