Intro to Python

Theresa Migler-VonDollen
CMPS 5P

Computers and Programs

- A modern computer can be defined as "a machine that stores and manipulates information under the control of a changeable program."
- Two elements:
  - Computers are devices for manipulating information.
  - Computers operate under the control of a changeable program.

Computers and Programs

- What is a computer program?
  - A detailed, step-by-step set of instructions telling a computer what to do.
  - If we change the program, the computer performs a different set of actions or a different task.
  - The machine stays the same, but the program changes.
  - Programs are executed, or carried out.

Programming

- All computers have the same power, with suitable programming, i.e. each computer can do the things any other computer can do.
- Software (programs) rule the hardware (the physical machine).
- The process of creating this software is called programming.

Programming

- Why learn to program?
  - Fundamental part of computer science
  - Having an understanding of programming helps you have an understanding of the strengths and limitations of computers.
  - Helps you become a more intelligent user of computers
  - It can be fun.
  - Helps the development of problem solving skills, especially in analyzing complex systems by reducing them to interactions between simpler systems.
  - Programmers are in great demand.

What is Computer Science?

- It is NOT the study of computers.
  - "Computers are to computer science what telescopes are to astronomy." - E. Dijkstra
- The question is: "What can be computed?"

What is Computer Science?

- Design
  - One way to show a particular problem can be solved is to actually design a solution.
  - This is done by developing an *algorithm*, a step-by-step process for achieving the desired result.
  - One problem - it can only answer the question of what can be computed in the positive. You can't prove a negative.

What is Computer Science?

- Analysis
  - Analysis is the process of examining algorithms and problems mathematically.
  - Some seemingly simple problems are not solvable by any algorithm. These problems are said to be unsolvable.
  - Problems can be intractable if they would take too long or take too much memory to be of practical value.

What is Computer Science?

- Experimentation
  - Some problems are too complex for analysis.
  - Implement a system and then study its behavior.

Hardware Basics

The central processing unit (CPU) is the "brain" of a computer.

- The CPU carries out all the basic operations on the data.
- Examples: simple arithmetic operations, testing to see if two numbers are equal.

Hardware Basics

Memory stores programs and data.

- CPU can only directly access information stored in main memory (RAM or Random Access Memory).
- Main memory is fast, but volatile, i.e. when the power is interrupted, the contents of memory are lost.
- Secondary memory provides more permanent storage: magnetic (hard drive, floppy), optical (CD, DVD)

Hardware Basics

Input devices

- Information is passed to the computer through keyboards, mice, etc.

Output devices

- Processed information is presented to the user through the monitor, printer, etc..

Hardware Basics

Fetch-Execute cycle

- First instruction retrieved from memory
- Decode the instruction to see what it represents
- Appropriate action carried out.
- Next instruction fetched, decoded, and executed.
- Repeat.

Programming Languages

Natural language has ambiguity and imprecision problems when used to describe complex algorithms.

- Programs expressed in an unambiguous, precise way using programming languages.
- Every structure in programming language has a precise form, called its syntax.
- Every structure in programming language has a precise meaning, called its semantics.

Programming Languages

A programming language is like a code for writing the instructions that the computer will follow.

- Programmers will often refer to their program as computer code.
- Process of writing an algorithm in a programming language often called coding.

Programming Languages

High-level computer languages

- Designed to be used and understood by humans

Low-level computer languages

- Computer hardware can only understand a very low level language known as machine language

Programming Languages

Add two numbers - Low-level

- Load the number from memory location 2001 into the CPU
- Load the number from memory location 2002 into the CPU
- Add the two numbers in the CPU
- Store the result into location 2003

In reality, these low-level instructions are represented in binary (1's and 0's)

Binary numbers

- The modern binary number system was discovered by Gottfried Leibniz in 1679.
- Counting in binary:
  - 0 - 0
  - 1 - 1
  - 2 - 10
  - 3 - 11
  - 4 - 100
  - 5 - 101
- What is 137 in binary?

Programming Languages

Add two numbers -High-level

- $c = a + b$
- This needs to be translated into machine language that the computer can execute.
- Compilers convert programs written in a high-level language into the machine language of some computer.

Programming Languages

- Interpreters simulate a computer that understands a high-level language.
- The source program is not translated into machine language all at once.
- An interpreter analyzes and executes the source code instruction by instruction.

Programming Languages

Compiling vs Interpreting

- Once program is compiled, it can be executed over and over without the source code or compiler. If it is interpreted, the source code and interpreter are needed each time the program runs.
- Compiled programs generally run faster since the translation of the source code happens only once.

Programming Languages

Compiling vs Interpreting

- Interpreted languages are part of a more flexible programming environment since they can be developed and run interactively
- Interpreted programs are more portable, meaning the executable code produced from a compiler for a Pentium won't run on a Mac, without recompiling. If a suitable interpreter already exists, the interpreted code can be run with no modifications.

Python

When you start Python, you will see something like >>>
>>> is called a *prompt* indicating that Python is ready for us to give it a command. These commands are called statements.

```
>>> print("Hello, world")
Hello, world
>>> print(2+3)
5
>>> print("2+3=", 2+3)
2+3= 5
>>>
```

Python

Usually we want to execute several statements together that solve a common problem. Use a function.

```
>>> def hello():
      print("Hello")
      print("Computers are Fun")

>>>
```

- The first line tells Python we are defining a new function called hello.
- The following lines are indented to show that they are part of the hello function.
- The blank line (hit enter twice) lets Python know the definition is finished.

Python

```
>>> def hello():
        print("Hello")
        print("Computers are Fun")

>>>
```

- Notice that nothing has happened yet! We've defined the function, but we haven't told Python to perform the function.
- A function is invoked by typing its name.

```
>>> hello()
Hello
Computers are Fun
>>>
```

Python

```
>>> hello()
Hello
Computers are Fun
>>>
```

- What are the "( )" for?
- Commands can have changeable parts called parameters that are placed between the ( )'s

Practice with Parameters

Write a Python function that takes in a number as a parameter and prints the square of that number.

Practice with Parameters

Write a Python function that takes in two numbers as parameters and prints the sum of the squares of the two numbers.

Python Programs

- When we exit the Python prompt, the functions we've defined cease to exist.
- Programs are usually composed of functions, modules, or scripts that are saved on disk so that they can be used again and again.
- A module file is a text file created in text editing software (saved as "plain text") that contains function definitions.
- A programming environment is designed to help programmers write programs and usually includes automatic indenting, highlighting, etc.

Python Programs

- We'll use filename.py when we save our work to indicate it's a Python program.
- In this code we're defining a new function called main.
- The main() at the end tells Python to run the code.

Practice with a Python program

Write a Python program that greets the user, and asks for an input number, call it x. Do the following 10 times:
$x = 3.9 * x * (1 - x)$.
Call this program chaos.py

Python comments

- Lines that start with # are called comments
- Intended for human readers and ignored by Python
- Python skips text from # to end of line

Python main function

```
def main():
```

- Beginning of the definition of a function called main
- Since our program has only this one module, it could have been written without the main function.
- The use of main is customary, however.

Python variables

```
x = eval(input("Enter a number between 0 and 1: "))
```

- *x* is an example of a *variable*
- A variable is used to assign a name to a value so that we can refer to it later.
- The quoted information is displayed, and the number typed in response is stored in x.

Python loops

```
for i in range(10):
```

- For is a *loop* construct
- A loop tells Python to repeat the same thing over and over.
- In this example, the following code will be repeated 10 times.

Python body

```
x = 3.9 * x * (1 - x)
print(x)
```

- These lines are the *body* of the loop.
- The body of the loop is what gets repeated each time through the loop.
- The body of the loop is identified through indentation.
- The effect of the loop is the same as repeating this two lines 10 times.

Python assignment

```
x = 3.9 * x * (1 - x)
```

- This is called an *assignment* statement
- The part on the right-hand side of the $=$ is a mathematical expression.
- $\star$ is used to indicate multiplication
- Once the value on the right-hand side is computed, it is stored back into (assigned) into x

Python execute main

```
main()
```

- This last line tells Python to execute the code in the function main

Python chaos.py program

```
def main():
    print("This program illustrates a chaotic function")
    x = eval(input("Enter a number between 0 and 1: "))
    for i in range(10):
        x = 3.9 * x * (1 - x)
        print(x)
main()
```

- For any given input, returns 10 seemingly random numbers between 0 and 1.
- It appears that the value of *x* is chaotic....

The Software Development Process

The process of creating a program is often broken down into stages according to the information that is produced in each phase.

- **Analyze the Problem** Figure out exactly the problem to be solved. Try to understand it as much as possible.
- **Determine Specifications** Describe exactly what your program will do.
  - Don't worry about how the program will work, but what it will do.
  - Includes describing the inputs, outputs, and how they relate to one another.

The Software Development Process

- **Analyze the Problem**
- **Determine Specifications**
- **Create a Design**
  - Formulate the overall structure of the program.
  - This is where the how of the program gets worked out.
  - You choose or develop your own algorithm that meets the specifications.
- **Implement the Design**
  - Translate the design into a computer language (Python).

The Software Development Process

- **Analyze the Problem**
- **Determine Specifications**
- **Create a Design**
- **Implement the Design**
- **Test/Debug the Program**
  - Try out your program to see if it worked.
  - If there are any errors (bugs), they need to be located and fixed. This process is called *debugging*.
  - Your goal is to find errors, so try everything that might "break" your program!

The Software Development Process

- **Analyze the Problem**
- **Determine Specifications**
- **Create a Design**
- **Implement the Design**
- **Test/Debug the Program**
- **Maintain the Program**
  - Continue developing the program in response to the needs of your users.
  - In the real world, most programs are never completely finished - they evolve over time.

The Software Development Process

- **Analyze the Problem**
- **Determine Specifications**
- **Create a Design**
- **Implement the Design**
- **Test/Debug the Program**
- **Maintain the Program**

Let's try it!

Temperature Converter

Write a Python program that asks the user for a temperature in Celcius and tells the user what the temperature is in Fahrenheit.

Temperature Converter

- **Analyze the Problem**
  - The temperature is given in Celsius, user wants it expressed in degrees Fahrenheit.
- **Determine Specifications**
  - Input - temperature in Celsius
  - Output - temperature in Fahrenheit
  - Output = 9/5(Input) + 32

Temperature Converter

- **Create a Design**
  - Input, Process, Output (IPO)
  - Prompt the user for input (Celsius temperature)
  - Process it to convert it to Fahrenheit using $F = 9/5(C) + 32$
  - Output the result by displaying it on the screen.
- Before we start coding, let's write a rough draft of the program in *pseudocode*.
- Pseudocode is precise English that describes what a program does, step by step.
- Using pseudocode, we can concentrate on the algorithm rather than the programming language.

Temperature Converter

- **Create a Design**
  - Pseudocode:
    - Input the temperature in degrees Celsius (call it celsius)
    - Calculate fahrenheit as (9/5)*celsius+32
    - Output fahrenheit

Temperature Converter

- **Implement the Design**
- **Test the Program**
  - Does the program work for all numbers? Decimals? Negative numbers?
- **Maintain the Program**
  - Do you anticipate any additions a user of this program might have?
  - Perhaps an extra check to see if the temperature is below freezing?

Elements of the Program

- Names
  - Names are given to variables (celsius, fahrenheit), modules (main, convert), etc.
  - These names are called identifiers
  - Every identifier must begin with a letter or underscore, followed by any sequence of letters, digits, or underscores.
  - Identifiers are case sensitive.
  - Some identifiers are part of Python itself. These identifiers are known as reserved words. This means they are not available for you to use as a name for a variable, etc. in your program.
  - and, del, for, is, raise, assert, elif, in, print, etc.
  - For a complete list, see table 2.1

Elements of the Program

- Expressions
  - The fragments of code that produce or calculate new data values are called expressions.
  - Literals are used to represent a specific value, e.g. 3.9, 1, 1.0
  - Simple identifiers can also be expressions.
  - Simpler expressions can be combined using operators.
  - $+, -, *, /, **$
  - Spaces are irrelevant within an expression.
  - The normal mathematical precedence applies.

Elements of the Program

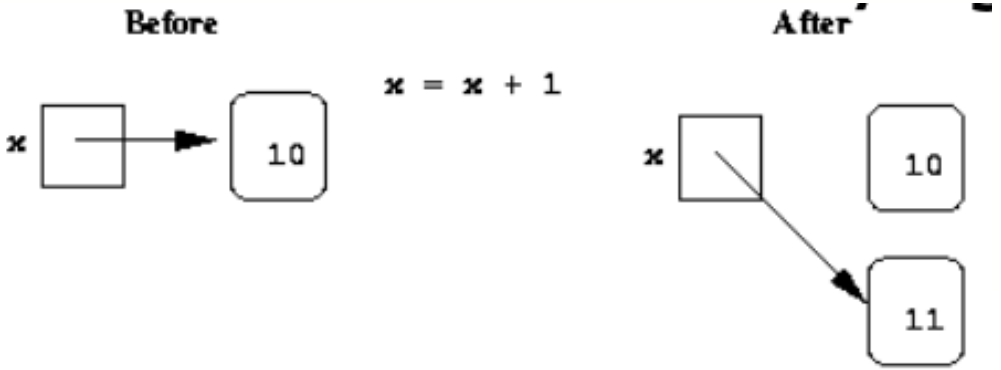
- Output Statements
  - A `print` statement can print any number of expressions.
  - Successive `print` statements will display on separate lines.
  - A bare `print` will print a blank line.

Elements of the Program

- Assignment Statements
  - Simple Assignment:
    <variable> = <expr>
    variable is an identifier, expr is an expression
  - The expression on the right-hand side is evaluated to produce a value which is then associated with the variable named on the left-hand side.
  - For example: fahrenheit = 9/5∗ celsius +32
  - Variables can be reassigned.
  - Variables are like a box we can put values in.
  - When a variable changes, the old value is erased and a new one is written in.
    NOT exactly

Elements of the Program

- Assignment Statements
  - Python doesn't overwrite these memory locations (boxes).
  - Assigning a variable is more like putting a sticky note on a value and saying, "this is x".



Elements of the Program

- Assignment Input
  - The purpose of an input statement is to get input from the user and store it into a variable.
  - $< variable >= eval(input(< prompt >))$
  - First the prompt is printed
  - The input part waits for the user to enter a value and press <enter>
  - The expression that was entered is evaluated to turn it from a string of characters into a Python value (a number).
  - The value is assigned to the variable.

Elements of the Program

- Simultaneous Assignment
  - Several values can be calculated at the same time.
  - $< var >, < var >, ... =< expr >, < expr >, ...$
  - Evaluate the expressions on the right-hand side and assign them to the variables on the left-hand side.
  - $sum, diff = x + y, x - y$
  - How could you use this to swap the values for x and y?
    - Why doesn't this work?
      $x = y$
      $y = x$
  - We could use a temporary variable...
  - But it's even easier in Python:
    $x, y = y, x$

Elements of the Program

- Definite Loops
  - A *definite loop* executes a definite number of times, i.e., at the time Python starts the loop it knows exactly how many iterations to do.
    - for <var> in <sequence>:
      <body>
  - The beginning and end of the body are indicated by indentation.
  - The variable after the for is called the *loop index*. It takes on each successive value in sequence.
  - `range` is a built-in Python function that generates a sequence of numbers, starting with 0.
  - `list` is a built-in Python function that turns the sequence into an explicit list.

Practice with Future Value Program

Write a Python program that takes an amount of money (the principal) and an apr and calculates the amount of the inverstment after 10 years.

Practice with Future Value Program

- **Analysis**
  - Money deposited in a bank account earns interest.
  - How much will the account be worth 10 years from now?
  - Inputs: principal, interest rate
  - Output: value of the investment in 10 years

Practice with Future Value Program

- **Specification**
  - User enters the initial amount to invest, the principal.
  - User enters an annual percentage rate, the interest.
  - The specifications can be represented like this:
    - *Program*: Future Value
      Inputs :
      principal: the amount of money being invested, in dollars
      apr: the annual percentage rate expressed as a decimal number.
      Output: The value of the investment 10 years in the future
      Relationship: Value after one year is given by $principal * (1 + apr)$. This needs to be done 10 times.

Practice with Future Value Program

- **Design**
  - Print an introduction
    Input the amount of the principal (principal)
    Input the annual percentage rate (apr)
    Repeat 10 times:
    $principal = principal * (1 + apr)$
    Output the value of principal

Practice with Future Value Program

- **Implementation**
  - Each line translates to one line of Python (in this case)
  - Print an introduction
    print ("This program calculates the future value")
    print ("value of a 10-year investment.")
  - Input the amount of the principal:
    principal = eval(input("Enter the initial principal: "))
  - Input the annual percentage rate
    apr = eval(input("Enter the annual interest rate: "))
  - Repeat 10 times: for i in range(10):
  - Calculate $principal = principal * (1 + apr)$
  - Output the value of the principal at the end of 10 years print ("The value in 10 years is:", principal)

Practice with Future Value Program

- **Debug**
- **Maintain**
  - Are there any immediate extensions or improvements that we can make to our program?
    - Calculate the value of the investment for a variable amount of years.
    - Allow for semianual compounding.

Numeric Data Types

- The information that is stored and manipulated by computer programs is referred to as *data*.
- In Python, there are two different kinds of numbers:
  - Whole numbers: 3, 7, 2094
  - Numbers with a decimal: .25, 45.10, 67.0525
- Inside the computer, whole numbers and decimal fractions are represented quite differently.
- We say that decimal fractions and whole numbers are two different data types.

Numeric Data Types

- The data type of an object determines what values it can have and what operations can be performed on it.
- Whole numbers are represented using the integer (int for short) data type.
- These values can be positive or negative whole numbers.
- Numbers that can have fractional parts are represented as floating point (or float) values.

Numeric Data Types

- How do we tell the two data types apart?
  - A numeric literal without a decimal point produces an int value.
  - A literal that has a decimal point is represented by a float (even if the fractional part is 0).
- Python's `type` function tell us what type the input is.

Determining Types

Write a Python function that takes any input and returns the type of that input.

- Notice that the input must be received as a parameter.

Numeric Data Types

- Why do we need two data types for numbers?
  - Values that represent counts can't be fractional, you can't loop 4.5 times.
  - Most mathematical algorithms are very efficient with integers
  - The float type stores only an approximation to the real number being represented.
  - Since floats aren't exact, use an int whenever possible.
  - Operations on ints produce ints, operations on floats produce floats (except for division).

Numeric Data Types

- Integer division produces a whole number.
- That's why $10//3 = 3$
- Think of it as how many times 3 goes into 10 where $10//3 = 3$ because 3 goes into 10 3 times with a remainder 1.
- $10\%3 = 1$ is the remainder of the integer division of 10 by 3.
- $a = (a/b)(b) + (a\%b)$

Using the Math Library

- Besides the usual arithmetic functions, there are many other math functions available in the math library.
- A *library* is a module with some useful definitions/functions.
- Suppose we wanted to compute the roots of a quadratic equation:
  $ax^2 + bx + c = 0$
- The square root function is in the math library.

Using the Math Library

- To use a library, we need to make sure this line is in our program:

  ```
  import math
  ```

- Importing a library makes whatever functions are defined within it available to the program.
- To access the sqrt library routine, we need to access it as `math.sqrt(x)`.
- Using this dot notation tells Python to use the `sqrt` function found in the math library module.
- To calculate the root:

  ```
  discRoot = math.sqrt(b*b - 4*a*c)
  ```

Practice

Write a program that asks the user for the coefficients of a quadratic equation and returns the real roots of the equation.

- What if the roots are imaginary?

Generating Random Numbers

- `random` is another useful library.
- `random.randint(x,y)` generates a random integer between (inclusive) `x` and `y`.
- Lets test to see if each number in the range is equally likely to be generated.

Write a program that generates 1000 random integers between 1 and 3 and counts the occurence of each number.

- Is this what we would expect?

Accumulating Results

- Say you are waiting in a line with five other people. How many ways are there to arrange the six people?
- 720 – 720 is the factorial of 6 (abbreviated 6!).
- Factorial is defined as: $n! = n(n-1)(n-2)...(1)$
- So, $6! = 6 * 5 * 4 * 3 * 2 * 1 = 720$
- How could we write a program to do this?

Accumulating Results

Write a Python program that takes an integer, x, and prints the factorial $x!$.

Accumulating Results

- How did we calculate 6!?
- Using repeated multiplications, and keeping track of the running product.
- This algorithm is known as an *accumulator*, because we're building up or accumulating the answer in a variable, known as the accumulator variable.
- The general form of an accumulator algorithm looks like this:
  Initialize the accumulator variable
  Loop until final result is reached update the value of accumulator variable

The `range` Function - Generalizing Factorials

- What if we want to generalize our factorial program?
- The `range(n)` function can do this.
- `range` has optional parameters:
  `range(start,n)`
  `range(start,n,step)`

Factorials Get Large - Fast

- What is 100!?
- That's huge, Python 3 can handle it, but previous versions and other languages cannot.
- While there are an infinite number of integers, there is a finite range of ints that can be represented.

Limits on Int

- This range of integers that can be represented depends on the number of bits a particular CPU uses to represent an integer value.
- Typical PCs use 32 bits.
- That means there are 232 possible values, centered at 0.
- This range then is -231 to 231-1. We need to subtract one from the top end to account for 0.
- But our 100! is much larger than this. How does it work?

Handling Large Numbers

- Does switching to float data types get us around the limitations of ints?
- If we initialize the accumulator to 1.0, what do we get?
- We no longer get an exact answer!
- Very large and very small numbers are expressed in scientific or exponential notation.
  $1.307674368e+012 = 1.307674368 \times 10^{12}$
- Here the decimal needs to be moved right 12 decimal places to get the original number, but there are only 9 digits, so 3 digits of precision have been lost.

Handling Large Numbers

- Floats are approximations.
- Floats allow us to represent a larger range of values, but with lower precision.
- Python has a solution, expanding ints.
- Python ints are not a fixed size and expand to handle whatever value it holds.
- Newer versions of Python automatically convert your ints to expanded form when they grow so large as to overflow.
- We get indefinitely large values (e.g. 100!) at the cost of speed and memory.

Type Conversions

- We know that combining an int with an int produces an int, and combining a float with a float produces a float.
- What happens when you mix an int and float in an expression?
  $x = 5.0 + 2$
- What do you think should happen?
- For Python to evaluate this expression, it must either convert 5.0 to 5 and do an integer addition, or convert 2 to 2.0 and do a floating point addition.

Type Conversions

- Converting a float to an int will lose information.
- ints can be converted to floats by adding ".0"
- In mixed-typed expressions Python will convert ints to floats.
- Sometimes we want to control the type conversion. This is called explicit typing.
- For example:
  - `int(4.5)`
  - `float(7)`
- We can also round using the `round` function.
  - What is the difference between `int(8.9)` and `round(8.9)`?

Fizz-buzz

- Write a program which prints out each number from 1 to 1,000.
- For all numbers divisible by 3 only print the word “fizz”, not the number.
- For all numbers divisible by 5 only print the word “buzz”, not the number.
- For all numbers divisible by 3 and 5 print only the word “fizzbuzz”.

The String Data Type

- The most common use of personal computers is word processing.
- Text is represented in programs by the string data type.
- A *string* is a sequence of characters enclosed within quotation marks or apostrophes.
- How do we get a string as input?

The String Data Type

- We can access the individual characters in a string through indexing.
- The positions in a string are numbered from the left, starting with 0.
- The general form is <string> [<expr>], where the value of expr determines which character is selected from the string.
- In a string of n characters, the last character is at position $n - 1$ since we start counting with 0.
- We can index from the right side using negative indexes.

The String Data Type

- Indexing returns a string containing a single character from a larger string.
- We can also access a contiguous sequence of characters, called a *substring*, through a process called *slicing*.
- Slicing: <string> [<start>:<end>]
- start and end should both be ints.
- The slice contains the substring beginning at position start and runs up to but doesn't include the position end.

The String Data Type

- Can we put two strings together into a longer string?
- Concatenation "glues" two strings together $+$.
- Repetition builds up a string by multiple concatenations of a string with itself $\star$.
- The function `len` returns the length of a string.

Simple String Processing

Write a Python program that asks the user for her first and last name and then prints a username. The username should be the first initial of the first name and the first seven characters of the last name.

- Can we make the username lowercase?
- Can we exclude punctuation?

Other String Methods

- There are a number of other string methods:
  - s.capitalize() - Copy of s with only the first character capitalized.
  - s.title() - Copy of s; first character of each word capitalized.
  - s.center(width) - Center s in a field of given width.
  - s.count(sub) - Count the number of occurrences of sub in s.
  - s.find(sub) - Find the first position where sub occurs in s.
  - s.join(list) - Concatenate list of strings into one large string using s as separator.
  - s.ljust(width) - Like center, but s is left- justified.

Other String Methods

- There are a number of other string methods:
  - s.lower() - Copy of s in all lowercase letters
  - s.lstrip() - Copy of s with leading whitespace removed
  - s.replace(oldsub, newsub) - Replace occurrences of oldsub in s with newsub
  - s.rfind(sub) - Like find, but returns the right-most position
  - s.rjust(width) - Like center, but s is right- justified
  - s.rstrip() - Copy of s with trailing whitespace removed
  - s.split() - Split s into a list of substrings
  - s.upper() - Copy of s; all characters converted to uppercase

Improved Username

Modify the username program so that the username is all lowercase with no punctuation.

- Hint: Use the `string` library and `string.punctuation`.

Simple String Processing

Write a Python program that asks for an int between 1 and 12 and returns the three letter abbreviation for the corresponding month.

- Hint: store all the names in one big string: "JanFebMarAprMayJunJulAugSepOctNovDec" and slice.

- One weakness - this method only works where the potential outputs all have the same length.
- How could you handle spelling out the months?

Strings, Lists, and Sequences

- It turns out that strings are really a special kind of sequence, so these operations also apply to sequences.
- Strings are always sequences of characters, but lists can be sequences of arbitrary values.
- Lists can have numbers, strings, or both.
- We can use the idea of a list to make our previous month program even simpler.
- We change the lookup table for months to a list.

Strings, Lists, and Sequences

- Lists are *mutable*, meaning they can be changed.
- Strings can not be changed.
- Inside the computer, strings are represented as sequences of 1's and 0's, just like numbers.
- A string is stored as a sequence of binary numbers, one number per character.
- It doesn't matter what value is assigned as long as it's done consistently.

Strings, Lists, and Sequences

- In the early days of computers, each manufacturer used their own encoding of numbers for characters.
- ASCII system (American Standard Code for Information Interchange) uses 127 bit codes.
- Python supports Unicode (100,000+ characters).
- The `ord` function returns the numeric (ordinal) code of a single character.
- The `chr` function converts a numeric code to the corresponding character.
- Using `ord` and `char` we can convert a string into and out of numeric form.

Strings, Lists, and Sequences

Write a "secret code" program.
For each character in a message print the corresponding number of the character.

- Hint: a `for` loop iterates over a sequence of objects, so the `for` loop looks like:
  `for ch in <string>`.

Strings, Lists, and Sequences

- Strings are objects that have useful methods associated with them.
- One of these methods is *split*. This will split a string into substrings based on spaces.
- Example:
  >>> “Hello string methods!”.split()
  (‘Hello’, ‘string’, ‘methods!’)
- Split can be used on characters other than space, by supplying the character as a parameter.

Strings, Lists, and Sequences

Now write a decoder.

- Hint: Convert a string containing digits into a number by using `eval`.
- Hint: Use a string accumulator variable, initialize as the empty string, `""`.

From Encoding to Encryption

- The process of encoding information for the purpose of keeping it secret or transmitting it privately is called *encryption*.
- Cryptography is the study of encryption methods.
- Encryption is used when transmitting credit card and other personal information to a web site.
- Strings are represented as a sort of encoding problem, where each character in the string is represented as a number that's stored in the computer.
- The code that is the mapping between character and number is an industry standard, so it's not "secret".

From Encoding to Encryption

- The encoding/decoding programs we wrote use a substitution cipher, where each character of the original message, known as the plaintext, is replaced by a corresponding symbol in the cipher alphabet.
- The resulting code is known as the *ciphertext*.
- This type of code is relatively easy to break.
- Each letter is always encoded with the same symbol, so using statistical analysis on the frequency of the letters and trial and error, the original message can be determined.

From Encoding to Encryption

- Modern encryption converts messages into numbers.
- Sophisticated mathematical formulas convert these numbers into new numbers - usually this transformation consists of combining the message with another value called the "key"
- To decrypt the message, the receiving end needs an appropriate key so the encoding can be reversed.

From Encoding to Encryption

- In a private key system the same key is used for encrypting and decrypting messages. Everyone you know would need a copy of this key to communicate with you, but it needs to be kept a secret.
- In public key encryption, there are separate keys for encrypting and decrypting the message.
- In public key systems, the encryption key is made publicly available, while the decryption key is kept private.
- Anyone with the public key can send a message, but only the person who holds the private key (decryption key) can decrypt it.

Input/Output as String Manipulation

Write a Python program that takes a date in numeric format and prints the same date in words.
For example: we want to enter a date in the format "05/24/2003" and print "May 24, 2003".

Input/Output as String Manipulation

- Sometimes we want to convert a number into a string.
- We can use the `str` function.
- If value is a string, we can concatenate a period onto the end of it.
- If value is an int, what happens?

Files: Multi-line Strings

- A *file* is a sequence of data that is stored in secondary memory.
- Files can contain any data type, but the easiest to work with are text.
- A file usually contains more than one line of text.
- Python uses the standard newline character (\n) to mark line breaks.

Files: Multi-line Strings

- Hello
  World

  Goodbye
- Stored in a file as:
  Hello\nWorld\n\nGoodbye\n

Files: Multi-line Strings

- The process of opening a file involves associating a file on disk with an object in memory.
- We can manipulate the file by manipulating this object.
  - Read from the file
  - Write to the file
- When done with the file, it needs to be closed. Closing the file causes any outstanding operations and other bookkeeping for the file to be completed.
- In some cases, not properly closing a file could result in data loss.

Files Processing

- Working with text files in Python:
  - Associate a disk file with a file object using the open function <filevar>=open(<name>, <mode>)
  - Name is a string with the actual file name on the disk. The mode is either 'r' or 'w' depending on whether we are reading or writing the file.
  - Infile = open("numbers.dat", "r")

Files Methods

- <file>.read() - returns the entire remaining contents of the file as a single (possibly large, multi-line) string.
- <file>.readline() - returns the next line of the file. This is all text up to and including the next newline character.
- <file>.readlines() - returns a list of the remaining lines in the file. Each list item is a single line including the newline characters.

Read in a Book

- Go to http://www.gutenberg.org, download a book as plain text, copy and paste it into a text editor, and save it as a .txt file.
- Write a Python program that reads in the text file and prints the word count and average word length.

Functions

We have seen 3 types of functions:

- Our programs that comprise a single function.
- Built-in Python functions (abs, round, $+$).
- Functions from standard libraries (math.sqrt).

Functions

- Having similar or identical code in multiple places has drawbacks:
  - Having to write the same code multiple times (time consuming).
  - The code must be maintained in multiple places.
- Functions can be used to reduce code duplication and make programs more easily understood and maintained.

Functions

- How to use functions:
  - Write a sequence of statements and then give that sequence a name.
  - Execute this sequence at any time by referring to the name.
- The part of the program that creates a function is called a function definition.
- When the function is used in a program, we say the definition is called or invoked.

Functions - Example

Write a program to sing the "happy birthday" song to Molly.

- There are a lot of duplicate print statements. We can make these duplicate statements into a little function named "happy".
- Now write a program to sing the "happy birthday" song to Andrew.
- There is still a lot of duplication. Perhaps use a parameter.

Functions and Parameters

- A function definition has the following form:
  def <name>(<parameters>):
      <body>
- The name of the function must be an identifier.
- parameters is a possibly empty list of variable names.
- Formal parameters, like all variables used in the function, are only accessible in the body of the function. Variables with identical names elsewhere in the program are distinct from the formal parameters and variables inside of the function body.

Functions and Parameters

- A function is called by using its name followed by a list of actual parameters or arguments: <name>(<actual-parameters>):
- When Python comes to a function call, it initiates a four-step process:
  - The calling program suspends execution at the point of the call.
  - The formal parameters of the function get assigned the values supplied by the actual parameters in the call.
  - The body of the function is executed.
  - Control returns to the point just after where the function was called.

Getting Results from a Function

- Passing parameters provides a mechanism for initializing the variables in a function.
- Parameters act as inputs to a function.
- We can call a function many times and get different results by changing its parameters.
- We've already seen numerous examples of functions that return values to the caller.
  `discRt = math.sqrt`$(b * b - 4 * a * c)$
- The value $b * b - 4 * a * c$ is the *actual parameter* of `math.sqrt`.
- We say `sqrt` **returns** the square root of its argument.

Simple Program with Return

Write a Python program that accepts two points (ordered pairs) as parameters and *returns* the Euclidean distance between those points.

Getting Results from a Function

- When Python encounters `return`, it exits the function and returns control to the point where the function was called.
- In addition, the value(s) provided in the return statement are sent back to the caller as an expression result.

Getting Results from a Function

- Sometimes a function needs to return more than one value.
- To do this, simply list more than one expression in the return statement.

Write a Python function that takes two numbers and returns their sum and product.

- When calling a function that returns more than one expression, use simultaneous assignment.

Getting Results from a Function

- Note: all Python functions return a value, whether they contain a return statement or not. Functions without a return hand back a special object, denoted `None`.
- A common problem is writing a value-returning function and omitting the `return`.

Functions that Modify Parameters

- The formal parameters of a function only receive the values of the actual parameters. The function does not have access to the variable that holds the actual parameter.
- Python is said to pass all parameters by *value*.
- Some programming languages (C++, Ada, and many more) do allow variables themselves to be sent as parameters to a function. This mechanism is said to pass parameters by *reference*.
- When a new value is assigned to the formal parameter, the value of the variable in the calling program actually changes.

Functions and Program Structure

- So far, functions have been used as a mechanism for reducing code duplication.
- Another reason to use functions is to make your programs more *modular*.
- As the algorithms you design get increasingly complex, it gets more and more difficult to make sense out of the programs.
- One way to deal with this complexity is to break an algorithm down into smaller subprograms, each of which makes sense on its own.

Data Collection

- Many programs deal with large collections of similar information.
  - Words in a document
  - Students in a course
  - Data from an experiment
- Recall the programming assignment where we asked the user to input grades and we returned the average.
  - That program didn't keep track of the actual numbers, just the sum.
  - What if we also wanted to compute the median?

Computing the Median

- The *median* is the data value that splits the data into equal-sized parts.
- For the data 1, 4, 5, 9, 42, the median is 5, since there are two values greater than 5 and two values that are smaller.
- One way to determine the median is to store all the numbers, sort them, and identify the middle value.

Computing the Median

- We need a way to store and manipulate an entire collection of numbers.
- Can we just use a lot of variables?
  - No, because we don't know how many we will need at the start.
- We need some way of combining an entire collection of values into one object.

Lists and Arrays

- Recall that Python lists are ordered sequences of items.
- A list or array is a sequence of items where the entire sequence is referred to by a single name and individual items can be selected by indexing.
- In other programming languages, arrays are generally a fixed size, meaning that when you create the array, you have to specify how many items it can hold.
- Arrays are generally also *homogeneous*, meaning they can hold only one data type.

Lists and Arrays

- Python lists are dynamic. They can grow and shrink on demand.
- Python lists are also *heterogeneous*, a single list can hold arbitrary data types.
- Python lists are mutable sequences of arbitrary objects.

Lists Operations

- Aside from all of the list operations that we have already seen, there is also the membership operation.
- `3 in ourList`
- Practice with:
  - `ourList.reverse()`
  - `ourList.sort()`
  - `ourList.count(2)`
  - `ourList.insert(5,'Thanks!')`
  - `ourList.remove(7)`

Lists Operations

- Most of these methods don't return a value - they change the contents of the list in some way.
- Lists can grow by appending new items, and shrink when items are deleted. Individual items or entire slices can be removed from a list using the `del` operator.

Statistics with Lists

Write a Python program that takes a list of numbers and calculates the mean and median.

- If the list has odd length, the middle value in the list is the median.
- If the list has even length, the median is the average of the middle two values.

Lists of Objects

- All of the list examples we've looked at so far have involved simple data types like numbers and strings.
- We can also use lists to store more complex data types.

Practice with lists

Write a Python function that takes a list and removes all duplicate values from the list.

Algorithms

Write a Python function that accepts a list of numbers and a number, x. If x is in the list, the function returns the position in the list where that number appears. If x is not in the list, the function returns the value -1.

- Note that this is very similar to the built in function `index`, except that it handles exceptions.
- However, we are interested in how the search is actually performed.

Linear Search

- Suppose that you are given a list of 1,000 numbers (unsorted) and you are asked to find a particular number.
  - How would you do this?
  - Scan the whole list.
- This strategy, scanning a list one by one, is called *linear search*.
- This strategy works well for modest-sized lists.
- Both `in` and `index` use linear searching algorithms.

Linear Search

- Now suppose that the list of number is sorted (low to high).
- Do we still need to scan the list one by one?
  - How would you do it?
  - You could scan until you find a number that is greater than x.
  - On average this will save half the time from our previous algorithm.

Binary Search

- How would you find x in a sorted list of 1,000 numbers?
  - You would probably look at the middle number in the list and compare it with x, then throw half the list away.
  - This strategy is called *binary search*. At each step, we divide the list into two parts and throw away one of the parts.

Binary Search

- Since x could be anywhere in the list, we start with `low` as the first location in the list and `high` as the last location in the list.
- This algorithm will look at the middle element in the range and compare it to x.
  - If x is larger than the middle element, we move `low` to be the location of the middle element.
  - If x is smaller than the middle element, we move `high` to be the location of the middle element.

Binary Search

Write a Python function, using binary search, that accepts a list of numbers and a number, x. If x is in the list, the function returns the position in the list where that number appears. If x is not in the list, the function returns the value -1.

Comparing Algorithms

- Which algorithm is better?
  - Which algorithm is easier to understand and implement?
  - Which algorithm runs faster?
- Intuitively, we might expect linear search to work well on small lists while binary search would work better on longer lists.
  - We could test *empirically*.
  - We could argue abstractly.

Comparing Algorithms

- To argue abstractly, we need to test the number of "steps".
- For searching, the difficulty is determined by the size of the collection - it takes more steps to find a number in a collection of a million numbers than it does in a collection of 10 numbers.
- How many steps are needed to find a value in a list of size n?
- In particular, what happens as n gets very large?

Comparing Algorithms

- Linear search:
  - If the list has size n, we would have to loop through at most n items.
  - The amount of time required is linearly related to the size of the list.
  - This is what computer scientists call a *linear time algorithm*.
- Binary search:
  - If the list has size n, we would need to loop through at most $\log_2(n)$ items.
  - This is what computer scientists call a *log time algorithm*.

Comparing Algorithms

- The logarithmic property can be very helpful.
- Suppose you have a New York City phone book with 12 million names.
  - You could walk up to a random New Yorker (that is listed in the phone book) and try guessing her name.
  - All you are allowed to do is ask if her name is alphabetically before or after one that you find in the phone book.
- How many guesses will you need?
  - $\log_2 12000000 \approx 24$
  - If you had used linear search it would be closer to 6 million!

Comparing Algorithms

- We mentioned earlier that Python uses linear search in its built-in searching methods. Why?
- Binary search required the data to be sorted.
- If the data is unsorted, it must first be sorted, then searched.

Recursive Problem-Solving

- The basic idea between the binary search algorithm was to successfully divide the problem in half.
- This technique is known as a *divide and conquer* approach.
- Divide and conquer divides the original problem into subproblems that are smaller versions of the original problem.
- In the binary search, the initial range is the entire list. We look at the middle element... if it is the target, we're done. Otherwise, we continue by performing a binary search on either the top half or bottom half of the list.

Recursive Problem-Solving

Write a Python function that calls itself to perform binary search.

- This version has no loop!

Recursive Problem-Solving

- A description of something that refers to itself is called a *recursive* definition.
- Unlike English, where you cannot use a word to define itself, in mathematics recursive definitions are common.
- An example of a recursive mathematical definition, revisit the factorial:
  - We know that $n! = n(n-1)(n-2)\ldots(3)(2)(1)$.
  - But we could also say: $n! = (n)((n-1)!)$

Recursive Problem-Solving

- Is this definition of factorial circular?
- No, because there is a base case: $1! = 1$

Write a Python function that recursively computes the factorial of a given number.

Recursive Problem-Solving

- Good recursive definitions have these two key characteristics:
  - There is one (or more) base case for which no recursion is applied.
  - All chains of recursion eventually end up at one of the base cases.
- The simplest way for these two conditions to occur is for each recursion to act on a smaller version of the original problem. A very small version of the original problem that can be solved without recursion becomes the base case.

Recursive Problem-Solving

- Python lists have built in methods that can be used to reverse a list.
- Suppose we wanted to reverse a string? recursively?

Write a recursive Python function that takes in a string as a parameter and returns the string with all characters in reversed order.

- Don't forget the base case! Here it is the empty string.

Anagrams

- An *anagram* is the result of rearranging the letters of a word or phrase to produce a new word or phrase, using all the original letters exactly once.
- An example is "silent" and "listen".
- Anagram formation is a special case of generating all permutations (rearrangements) of a sequence, a problem that is seen frequently in mathematics and computer science.

Anagrams

Write a Python function that, given a word, returns a list with all possible permutations of that word.

- Hint: Suppose we already had all permutations of the word without the first letter. How could we use this to find all permutations of the whole word?
- How many permutations of a word of length n are there?

Recursion versus Iteration

- There are similarities between iteration (looping) and recursion.
- In fact, anything that can be done with a loop can be done with a simple recursive function.
- Some problems that are simple to solve with recursion are quite difficult to solve with loops.
- Often, recursive solutions are more efficient than their iterative counterparts. But not always...

Fibonacci Numbers

$1, 1, 2, 3, 5, 8, 13, 21, \ldots$

Write an iterative Python program that takes an integer, x, as an input and returns a list with the first x Fibonacci numbers.

Write a recursive Python program that takes an integer, x, as an input and returns a list with the first x Fibonacci numbers.

Recursion versus Iteration

- Which program runs faster?
- The recursive algorithm is extremely inefficient because it performs many duplicate calculations.
- Recursion is another tool in your problem-solving toolbox.
- Sometimes recursion provides a good solution because it is more elegant or efficient than a looping version.
- At other times, when both algorithms are quite similar, the edge goes to the looping solution on the basis of speed.

Sorting Algorithms

- Suppose you are given a list of numbers in arbitrary order and your task is to rearrange them in increasing order.
- How would you do this?
- One simple (and correct) method is to look through the entire list and find the smallest element, then look through the entire remaining portion of the list for the second smallest element....
  - This algorithm is called *selection sort*.
  - Is this an efficient algorithm?

Selection Sort

Write a Python program that takes as an input a list of numbers and returns the same list in sorted (increasing) order using the selection sort algorithm.

Mergesort

- Suppose that you and your friend needed to sort a deck of cards quickly.
- You might split the deck in half, and each sort your half, then combine the two decks.
- This is a divide and conquer approach.
- This algorithm is called *mergesort*.
- Practice with an example on the board.

Mergesort

Write a Python program that correctly *merges* two sorted lists.

Write a Python program that takes as an input a list of numbers and returns the same list in sorted (increasing) order using the mergesort algorithm.

Selection Sort versus Mergesort

- Which algorithm should we use?
- Certainly, the time it takes to sort the list depends on the size of the list.
- The question is: How many steps does each algorithm require as a function of the size of the list being sorted?
- Suppose that we have n items in our list.

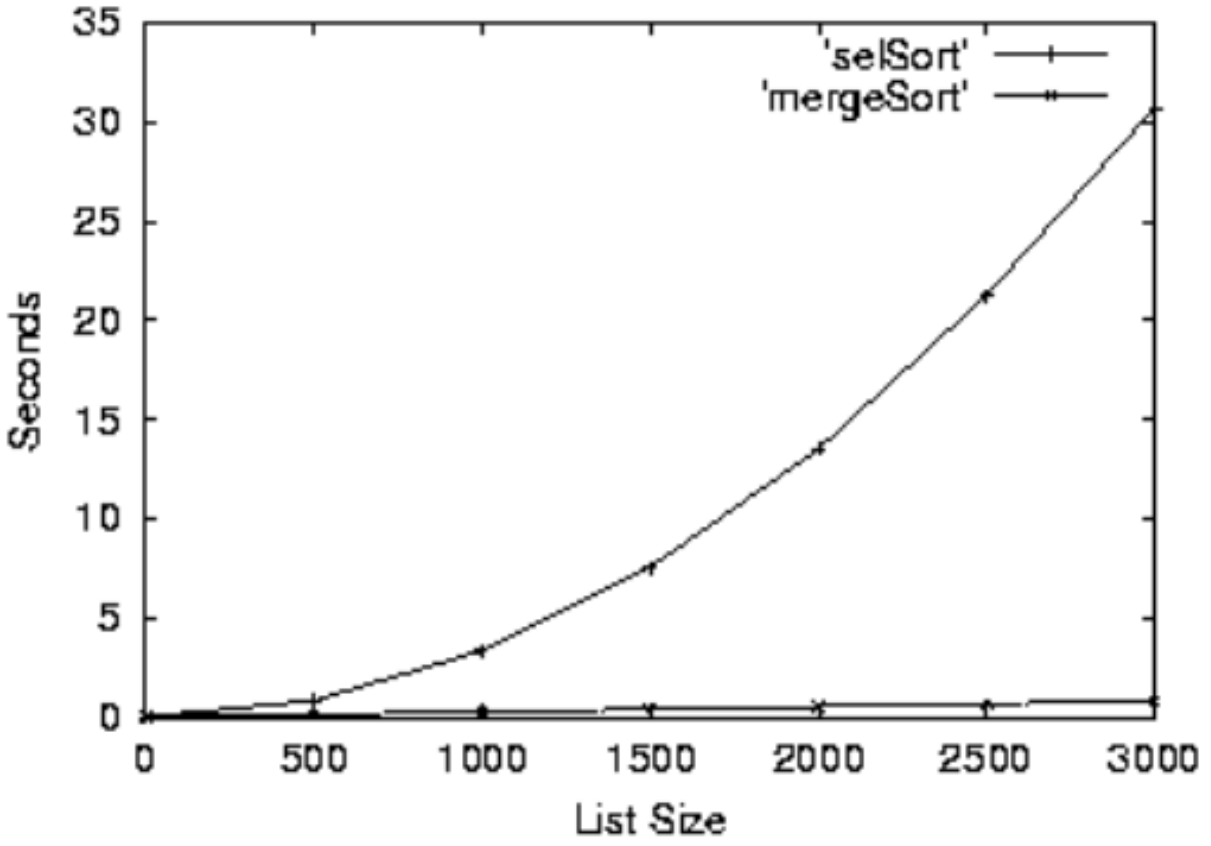
Running Time Analysis of Selection Sort

- For the first step, in the worst case, we need to look through all n items.
- For the second step, we might need to look through all remaining $n - 1$ items.
- The total number of iterations is:
  $n + (n-1) + (n-2) + (n-3) + \cdots + 2 + 1 = \frac{n(n+1)}{2}$
- So the running time is proportional to n^2.
- We call this a *quadratic* algorithm.

Running Time Analysis of Mergesort

- For Mergesort, the place where the real sorting occurs is in the *merge* function.
- Practice sorting $[3, 1, 4, 1, 5, 9, 2, 6]$.
- Starting from the top, we have to copy n values into the second level.
- From the second to the third levels, the n values need to be copied again.
- Each level requires copying n values....How many levels are there?
  - $\log_2 n$.
- Therefore the total work required is $n \log n$. This is called an $n \log n$ algorithm.

Comparing Quadratic and $n \log n$ Algorithms



Hard Problems

- There are some problems for which there is no *known* fast algorithm.
- The best we can seem to do on some problems is an *exponential* algorithm.
- What do computer scientists do when they encounter such problems?