HOW DOES COMPUTER PROGRAMMING WORK?

MAGIC.
Basic Concepts

• Computer Science
  • Computer -
  • Science -
• Programming history
• Algorithms
• Pseudo code
Basic Concepts

- Computer Science
  - Computer – a machine for performing calculations
  - Science –
- Programming history
- Algorithms
- Pseudo code
Basic Concepts

• Computer Science
  • Computer – a machine for performing calculations
  • Science – knowledge and study of a subject
• Programming history
• Algorithms
• Pseudo code
Computer Components

Software

User Programs

Operating System

Hardware
What are Programs?

- Programs provide instructions for computers
- Similar to giving directions to a person who is trying to get from point B from point A. A program may say... if starting at location B, go north 3 blocks, then go East 2 blocks.
Programming History

• Programming language tree:
  
  http://people.mandriva.com/~prigaux/language-study/diagram.png

• First programmer:
  • Ada Lovelace

• Different paradigms
  • Functional
  • Procedural
  • Object Oriented
  • etc.
What are Programs?

- Input
- Processing
- Output
High/Low Level Programming Languages

- Low level programming languages are languages which have instruction sets that are limited and specific to the computer hardware being run on. Programmers need to know how the hardware works to use it.

- High level programming languages provide a layer of abstraction that allows for the programmer to only have to learn a hardware independent language to write software.
Low Level Languages

- **Pros**
  - Can be more optimized
  - Usually smaller executables

- **Cons**
  - Platform dependent
  - Slower development
  - Harder to understand
  - Easier to introduce bugs

- When to use it: Need extra optimization, Need small executables (embedded systems)

- Example: Assembly (different for each OS/Architecture)
High Level Languages

- **Pros**
  - Faster development (libraries/etc.)
  - Easier to understand (abstraction)
  - Can be platform independent

- **Cons**
  - Usually larger executables
  - Abstraction layer usually adds overhead for processing resulting in slower executables

- **When to use it:** Need portability, need faster turn-around time for development, when programmer time is more valuable than processing time

- **Examples:** Java, C, C++, C#, Javascript, PHP, Perl, Python, Lisp, Scheme, R, etc.
Assembly Example (Low Level)

```assembly
mov ax, cs
mov ds, ax
mov ah, 9
mov dx, offset Hello
int 21h
xor ax, ax
int 21h
Hello:
    db "Hello World!", 13, 10, "$"
```
Java Example (High Level)

class HelloWorld {
    static public void main( String args[] ) {
        System.out.println( "Hello World!" );
    }
}


Python Example (High Level)

print(“hello World!”)
Anatomy of a Program
Bytecode vs. Executable

• Bytecode
  • Cross platform
  • Allows for replacement of small components without recompiling entire programs
  • Generally slower performance

• Executable
  • Runs on one platform
  • Programs generally compile down to larger executables
  • Generally faster performance
Interpreter

• In addition to bytecode, python provides a real time interpreter. An interpreter allows you to interactively provide statements and have python compile and run those statements on the fly.

• Provides
  • fast and user friendly development environment
  • easy way to debug programs
  • no waiting around for code to compile.
Natural Languages

- Syntax: punctuation and spacing
- Grammar: forms well defined sentences (e.g. subject–verb–object)
- Parts of speech: nouns, verbs, adjectives
- Semantics: The meaning of the words

Example: The quick brown fox jumped over the lazy dog.
Programming Languages

• Syntax: punctuation (e.g. parentheses, colons, spacing)

• Grammar: forms well defined statements (e.g. if statement is true then perform subsequent statement)

• Parts of speech: keywords, variables, operators, etc.

• Semantics: The meaning of the words

• Example: if a == b: print(“a is equal to b”)
Coding Style

- Just as in natural languages the style of coding matters.
- Style determines the level of readability, maintainability, and efficiency.
- Several things make up a coding style:
  - Formatting, Naming schemes, Comments, and more!
Algorithms
Algorithms

• A series of repeated instructions that solve a problem
Algorithms

- A series of repeated instructions that solve a problem
- For example:
  - To read a file and print its contents a program would: check if it has reached the end of a file, if not read a line and print its contents, repeat
  - To sort a list of numbers it could: compare the first two number, swap the number if necessary, then proceed to second and third numbers and repeat until all numbers are sorted
Bugs
Bugs

• All programmers run into bugs... lots and lots of bugs. Bugs are errors in a program.

• Bugs come in three different varieties:
  •
  •
  •
Bugs

• All programmers run into bugs... lots and lots of bugs. Bugs are errors in a program.

• Bugs come in three different varieties:
  • Compile time errors: (syntax/type/etc.) The rules of the language have been violated. Examples: improper spacing, missing colon, etc.
Bugs

• All programmers run into bugs... lots and lots of bugs. Bugs are errors in a program.

• Bugs come in three different varieties:
  • Compile time errors: (syntax/type/etc.) The rules of the language have been violated. Examples: improper spacing, missing colon, etc.
  • Runtime errors (crashes): Errors during execution Example: input file may not be readable
Bugs

• All programmers run into bugs... lots and lots of bugs. Bugs are errors in a program.

• Bugs come in three different varieties:
  • Compile time errors: (syntax/type/etc.) The rules of the language have been violated. Examples: improper spacing, missing colon, etc.
  • Runtime errors (crashes): Errors during execution Example: input file may not be readable
  • Logic errors: A problem that is caused by a flawed algorithm or set of instructions used to solve a problem. Example: If you try to print the numbers 1-10 using 'range(10)' instead of 'range(1,11)'
Pseudo-Code

• The expression of programming logic in a language independent nature.

• Good for design phase of coding

• Examples:

  if student_grade >= 60
    print “passed”
  else
    print “failed”
Writing a Program

1. Design
2. Write Code
3. Fix Syntax Errors
4. Run Program
Basic Programming Concepts

- Keywords
- Statements
- Operators
- Data Types
- Variables
- Assignment
- Constants
Python Keywords

- Keywords are reserved words that have special meaning in a particular programming language. These words cannot be used for any other purpose (e.g., variable names). Python keywords are:

  False, class, finally, is, return, None, continue, for, lambda, try, True, def, from, nonlocal, while, and, del, global, not, with, as, elif, if, or, yield, assert, else, import, pass, break, except, in, raise
Statements

• Programs are made up of a series of statements
• Similar to a sentence in a natural language, each statement presents a command that the interpreter understands
• The interpreter parses or reads the statement determines what is being requested and executes that statement
Data Types

- Integers (Real numbers): int
- Strings (Text): str
- Floating point numbers (decimals): float
Operators

- Different data types have different operators
- Operators can be part of a statement
- Operators act on the operands around them
- Unary operators (-3), Binary operators (1-3), Ternary operators (True if a<b else False)
<table>
<thead>
<tr>
<th>Operators on Numbers (in order of precedence)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Addition</strong></td>
</tr>
<tr>
<td>+</td>
</tr>
<tr>
<td><strong>Subtraction</strong></td>
</tr>
<tr>
<td>-</td>
</tr>
<tr>
<td><strong>Multiplication</strong></td>
</tr>
<tr>
<td>*</td>
</tr>
<tr>
<td><strong>Division</strong></td>
</tr>
<tr>
<td>/</td>
</tr>
<tr>
<td><strong>Division with integers</strong></td>
</tr>
<tr>
<td>//</td>
</tr>
<tr>
<td><strong>Remainder</strong></td>
</tr>
<tr>
<td>%</td>
</tr>
<tr>
<td><strong>Exponents</strong></td>
</tr>
<tr>
<td>** **</td>
</tr>
</tbody>
</table>
Operators on Strings (str)

Concatenation  +
Repetition      *
Comments

• Provides:
  • Documentation
  • Clarifying what specific code is doing
  • Make code easy for the author or other programmers to understand

• Line comments start with a '#' character

• Block comments start and end with either three single quotes ('') or three double quotes (""")
Comments Example

# list number from 0 to 9
list(range(10))

"""This code will list number from -10 to -1"
list(range(-10,0))

""""" This code will list numbers from -5 to 4"
list(range(-5,5))
Variables Names

• Variables are labels used by programmers for storage of data

• Variable name in python:
  • Can be long
  • Are case sensitive (Alpha != alpha)
  • Can contain letters, numbers, and underscores (_)
  • Must not start with a number
  • Can not be a keyword
Variables Names (Examples)

thisVariableNameIsValid = "valid naming conventions"
Variables Names (Examples)

thisVariableNameIsValid = "valid naming conventions"
this_variable_name_is_valid = "also valid"
Variables Names (Examples)

thisVariableNameIsValid = "valid naming conventions"
this_variable_name_is_valid = "also valid"
names = "names are valid"
Variables Names (Examples)

thisVariableNameIsValid = "valid naming conventions"
this_variable_name_is_valid = "also valid"
names = "names are valid"
Names = "not the same as names"
Variables Names (Examples)

thisVariableNameIsValid = "valid naming conventions"
this_variable_name_is_valid = "also valid"
names = "names are valid"
Names = "not the same as names"
numbers_between_0_and_9 = "another valid"
Variables Names (Examples)

thisVariableNameIsValid = "valid naming conventions"
this_variable_name_is_valid = "also valid"
names = "names are valid"
Names = "not the same as names"
numbers_between_0_and_9 = "another valid"
0_9_can_not_start_a_name = "invalid"
Variables Names (Examples)

thisVariableNameIsValid = "valid naming conventions"
this_variable_name_is_valid = "also valid"
names = "names are valid"
Names = "not the same as names"
numbers_between_0_and_9 = "another valid"
0_9_can_not_start_a_name = "invalid"
if = "keywords are not allowed"
Variables Names (Examples)

thisVariableNameIsValid = "valid naming conventions"
this_variable_name_is_valid = "also valid"
names = "names are valid"
Names = "not the same as names"
numbers_between_0_and_9 = "another valid"
0_9_can_not_start_a_name = "invalid"
if = "keywords are not allowed"
special_chars_!_allowed = "invalid"
Constants

- A constant is a stored value that doesn't change
- Used for things that will remain constant throughout the program. (e.g. pi, conversions between metric and standard units, etc)
- When representing data that doesn't change it's often a good idea to use constants
- In python, there is no such thing as a constant, but it's still good coding practice to use certain variables as constants
- Generally constants are in all CAPITAL letters
Variable Assignment Statements

$\ PI = 3.14159265$

$\ \text{radius} = 5$

$\ \text{area\_circle} = \ PI \ast (\text{radius}**2)$
Data Input

• For strings:
  name = input("What is your name? ")
  print("Hello, " + name)

• For numbers:
  year = input("What year is your birthday? ")
  year = int(year)  # convert string to integer
  print("You were born in " + str(year) + "?!")
  print("I have met many people born in " + str(year-1) + " and " + str(year+1) + " but never " + str(year))
Type Conversion

- In certain circumstances, your data types may need to be converted to other data types (e.g., converting a string into an integer or vice versa).

- **String → Integer**
  
  ```
  x = '14'
  int(x)  or  float(x)
  ```

- **Integer or Floats → String**
  
  ```
  x = 14
  str(x)
  ```

- **Integer->Float (done by Python)**
  
  `2.5*12`  is equivalent to  `2.5 * float(12)`
Data Output

- Print built-in statement
- Print takes any number of arguments
  ```python
p = ['jack', 'and', 'jill']
print(*p)
print('went up the hill')
```
- Print takes two special arguments at the end of the list of arguments:
  ```python
print('to', 'fetch', 'a', 'pail', sep = '.')
print('to', 'fetch', 'a', 'pail', end = '.')
print('to', 'fetch', 'a', 'pail', sep = '.', end = '
')
p = ['to', 'fetch', 'a', 'pail', end = '.', sep = '
')
```
Data Output

- For numbers with strings:
  
  ```python
  year = 2010
  print('The year is ', year)
  ```
For numbers with strings:

```python
year = 2010
print('The year is ', year)  # OK
print('The year is ' + year)
```
Data Output

• For numbers with strings:

```python
year = 2010
print('The year is ', year)    # OK
print('The year is ' + year)   # TypeError
print('The year is ' + str(year)) # OK
```