ECE12: Introduction to Programming
Review of Lectures 1-7

Rainer Dömer

doemer@uci.edu

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

Review of Lectures 1-7

- Lecture 1: Introduction, getting started
- Lecture 2: Introduction to Unix, Python
- Lecture 3: Python types, operators
- Lecture 4: Python statements, programs
- Lecture 5: Control structures, selection
- Lecture 6: Control structures, repetition
- Lecture 7: Functions

Unix System Environment

- Unix system commands
  - **`echo`** print a message
  - **`date`** print the current date and time
  - **`ls`** list the contents of the current directory
  - **`cat`** list the contents of files
  - **`more`** list the contents of files page by page
  - **`pwd`** print the path to the current working directory
  - **`mkdir`** create a new directory
  - **`cd`** change the current directory
  - **`cp`** copy a file
  - **`mv`** rename and/or move a file
  - **`rm`** remove (delete) a file
  - **`rmdir`** remove (delete) a directory
  - **`man`** view manual pages for system commands

Introduction to Programming

- Categories of programming languages
  - Machine languages (stream of 1's and 0's)
  - Assembly languages (low-level CPU instructions)
  - High-level languages (high-level instructions)
- Translation of high-level languages
  - Interpreter (translation for each instruction)
  - Compiler (translation once for all code)
  - Hybrid (combination of the above)
- Types of programming languages
  - Functional (e.g. Lisp)
  - Structured (e.g. Pascal, C, Ada)
  - Object-oriented (e.g. C++, Java, Python)

Introduction to Python Programming

- Python interpreter
  - interactive mode
    - like an advanced calculator
  - batch mode
    - program execution
- Basic data types
  - string “This is a string”, ‘This one, too!’
  - integer ..., -3, -2, -1, 0, 1, 2, 3, ...
  - floating point 12.34, 3.1415, 4.5e+8

Introduction to Python Programming

- Arithmetic operations
  - shift left, shift right <<, >>
  - addition, subtraction +, -
  - multiplication, division *, /
  - integer division, modulus //, %
  - exponentiation **
- Python programming, I/O
  - print formatted output (to stdout)
  - raw_input() string input (from stdin)

Our first Python Program

- Program file **`hello.py`**
  - **`# comment`** (until end of the line)
  - **`print`** function: formatted output (to stdout)

```
# hello.py: our first Python program
#
# author: Rainer Doemer
#
# modifications:
# 01/13/04 RD  initial version

print "Hello World!"
```

- Execute the program
  - run Python interpreter in batch mode
  - **`python hello.py`**
  - **`Hello World!`**
- Program modification
  - multiple statements...
  - text formatting using escape sequences...

Our first Python Program

- Text formatting using escape sequences
  - `\n` new line
  - `\t` horizontal tab
  - `\r` carriage return
  - `\b` back space
  - `\a` alert / bell
  - `\\` backslash character
  - `\”` double quote character
  - `\’` single quote character

String formatting

- String formatting operator **%**
  - % conversion specifiers in string (left argument) are replaced with formatted values (right argument)
  - Example: **`print "%s is %d years old." % ("Sophie",7)`**
  - Conversion specifiers
    - **%c** single ASCII character
    - **%s** string value (opt.: string length)
    - **%d** signed decimal integer (opt. number of digits)
    - **%u** unsigned decimal integer (opt. number of digits)
    - **%o** unsigned octal integer (opt. number of digits)
    - **%x,%X** unsigned hexadecimal integer (0-1a-f, 0-1A-F)
    - **%f** floating point number
    - **%e,%E** floating point number in scientific notation
    - **%g,%G** floating point number using least-significant digits
  - Optional formatting arguments
    - **-** left/right justification
    - **N** field width (i.e. number of digits/characters)
- String concatenation operator **+**
- String multiplication operator *****

Objects and Variables

- Objects are used to store data
- Every object has
  - a type (e.g. integer, floating point, string)
  - a value (e.g. 42, 3.1415, “text”)
  - a size (number of bytes in the memory)
  - a location (address in the memory, aka. identity)
- Objects are either
  - mutable (object value can be changed)
  - immutable (object value cannot be changed)
- Variables
  - serve as identifiers for objects
  - are bound to objects
  - give objects a name

Arithmetic Operations

- Evaluation order of expressions
  - left to right (except for exponentiation!)
  - by operator precedence:
    - unary plus, minus +, -
    - exponentiation **
    - multiplication, division, modulo *, /, %
    - addition, subtraction +, -
    - shift left, shift right <<, >>
    - bitwise and &
    - bitwise xor ^
    - comparison <, <=, ==, >=, >, !=, <>
    - logical not not
    - logical and and
    - logical or or

Relational Operators

- Relational operators (comparison of values)
  - **<** less than
  - **>** greater than
  - **<=** less than or equal to
  - **>=** greater than or equal to
  - **==** equal to (remember, **=** means assignment!)
  - **!=**, **<>** not equal to
- Comparison is defined for many types
  - integer (e.g. 5 < 6)
  - floating point (e.g. 7.0 < 7e1)
  - string (e.g. “alpha” < “beta”)
- Result type is boolean, but represented as an integer
  - false **0**
  - true **1**

Logical Operators

- Logical operators
  (often used together with relational operators)
  - **`not`** logical negation
  - **`and`** logical and
  - **`or`** logical or

x	y	not x	x and y	x or y
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

- Argument and result types are boolean, represented as integer (or other type)
  - false **`0`** (or zero **`0.0`**, empty string **`""`**, ...)
  - true **`1`** (or non-zero, non-empty string, ...)

Augmented Assignment Operations

- Assignment operator: **=**
  - evaluates right-hand side
  - assigns result to left-hand side
- Augmented assignment operator: **+=**, ***=**, ...
  - evaluates right-hand side as temp. result
  - applies operation to left-hand side and temp. result
  - assigns result to left-hand side
- Example: Counter
  - `x = 0` # initialization
  - `x = x + 1` # counting by regular assignment
  - `x += 1` # counting by augmented assignment
- Augmented assignment operators:
  - **+=**, **-=**, ***=**, **/=**, **%=**, ****=**, **<<=**, **>>=**, **^=**, **|=**, **&=**

Programming Principles

- Thorough understanding of the problem
- Problem Definition
  - Input data
  - Output data
- *Algorithm*: Procedure to solve the problem
  - Detailed set of *actions* to perform
  - Specification of *order* in which to perform the actions
  - Termination after a *finite* number of steps
- Pseudo code: Planning a program
  - Informal (English) description of steps in an algorithm
  - Example: Cake baking recipe
- Program: Instructions for the computer
  - Formal description in programming language
    - Statements (steps, actions)
    - Control structures (flow of control)
- Control flow
  - Execution order of statements in the program

Python Keywords

- Keywords in Python
  - **and**
  - **assert**
  - **break**
  - **class**
  - **continue**
  - **def**
  - **del**
  - **elif**
  - **else**
  - **except**
  - **exec**
  - **finally**
  - **for**
  - **from**
  - **global**
  - **if**
  - **import**
  - **in**
  - **is**
  - **lambda**
  - **not**
  - **or**
  - **pass**
  - **print**
  - **raise**
  - **return**
  - **try**
  - **while**
- These keywords are reserved and cannot be used as identifiers!
- More keywords may be used in future versions of the language

Block Indentation

- Python groups statements into blocks by use of indentation
  - Other languages typically use
    - parentheses ( ) e.g. Lisp
    - braces { } e.g. C, C++, Java
    - keywords begin end e.g. Pascal
- Example:

```
# some statements...
if x < 0:
    print x, “is negative!”
    # handle negative values of x...
    if x < 100:
        print x, “is too small!”
        # handle the problem
if x > 0:
    # handle positive values of x...
# more statements...
```

- Indentation increases readability of the code
  - in Python, proper indentation is required
  - in other languages, proper indentation is recommended

Block Indentation

- Python groups statements into blocks by use of indentation
  - Other languages typically use
    - parentheses ( ) e.g. Lisp
    - braces { } e.g. C, C++, Java
    - keywords begin end e.g. Pascal
- Example:

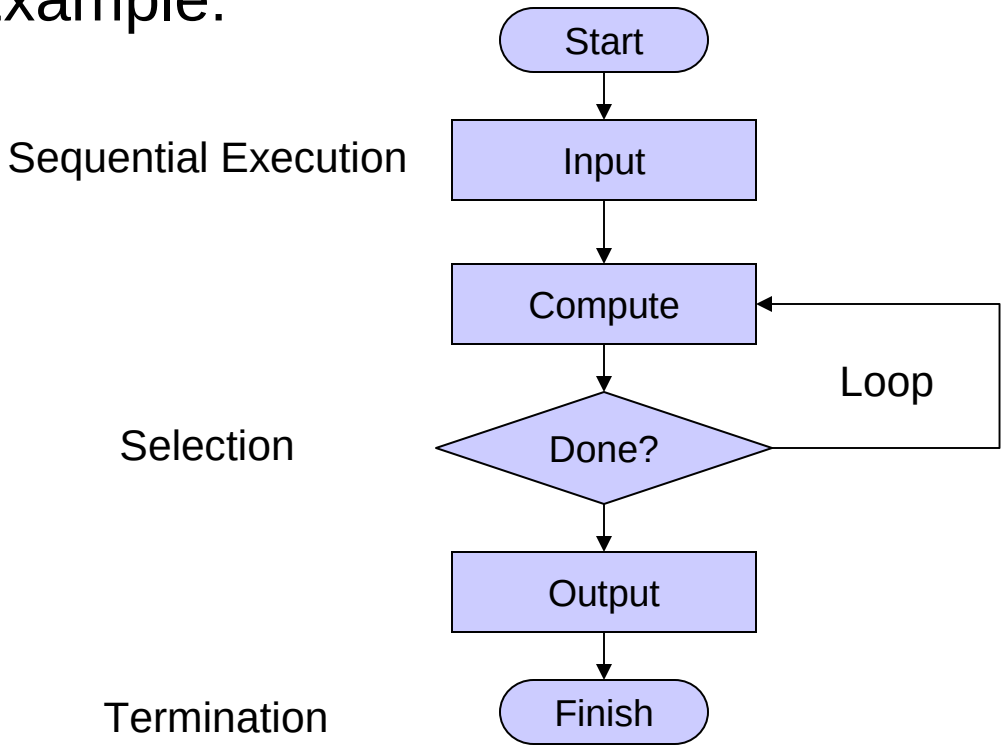
indentation level 0
indentation level 1
indentation level 2
indentation level 0
indentation level 1
indentation level 0

```
# some statements...
if x < 0:
    print x, “is negative!”
    # handle negative values of x...
    if x < 100:
        print x, “is too small!”
        # handle the problem
if x > 0:
    # handle positive values of x...
# more statements...
```

- Indentation increases readability of the code
  - in Python, proper indentation is required
  - in other languages, proper indentation is recommended

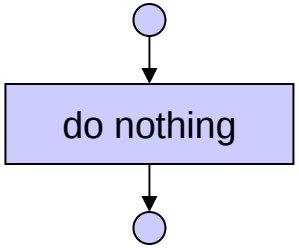
Control Structures

- Flow Charts
  - Graphical representation of program control flow
  - Example:



Empty Blocks

- **`pass`** statement
  - does nothing (no operation, no-op)
  - can be used to represent an empty block
  - Flow chart

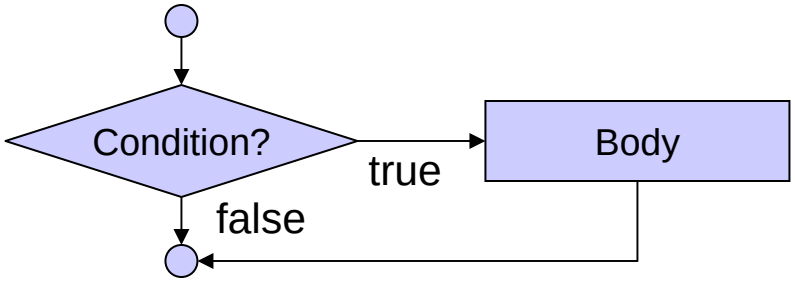


Example:

```
if grade >= 90:
    print “Ask for raise!”
elif grade >= 60:
    pass
else:
    print “Take class again!”
```

Selection Structures

- **`if`** statement
  - Flow chart:

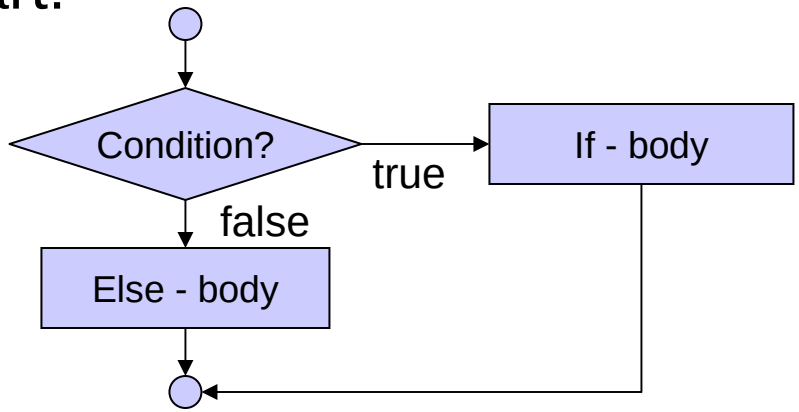


  - Example:

```
if grade >= 60:
    print “Passed.”
```

Selection Structures

- **`if` – `else`** statement
  - Flow chart:

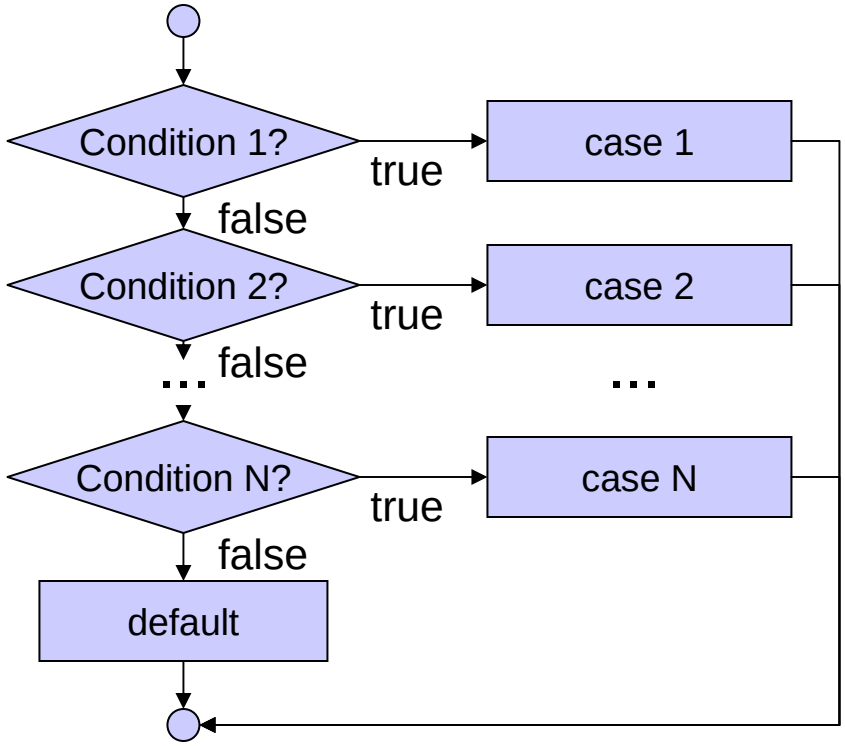


  - Example:

```
if grade >= 60:
    print “Passed.”
else:
    print “Failed.”
```

Selection Structures

- **`if` – `elif` – `else`** statement
  - Flow chart:

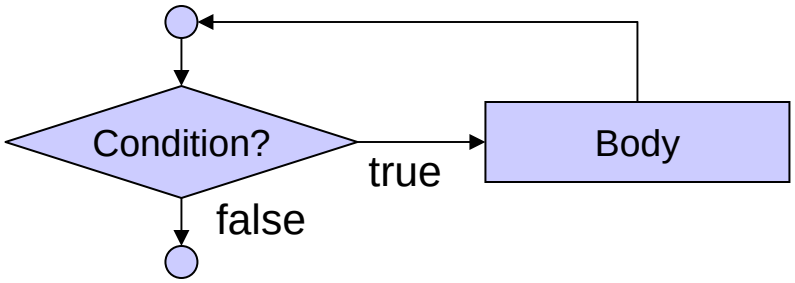


Example:

```
if grade >= 90:
    print “A”
elif grade >= 80:
    print “B”
elif grade >= 70:
    print “C”
elif grade >= 60:
    print “D”
else:
    print “F”
```

Control Structures

- **`while`** loop
  - Repetition structure (iteration)
  - Flow chart:

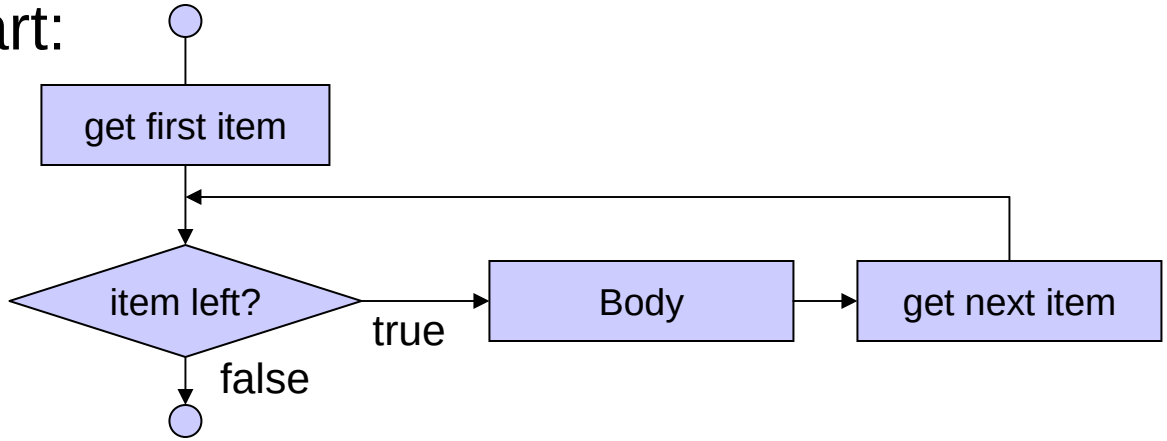


  - Example:

```
product = 2
while product <= 1000:
    product *= 2
print product
```

Control Structures

- **`for`** loop
  - Repetition structure (iteration over lists)
  - Flow chart:



  - Example:

```
for name in [“Alan”, “Bob”, “Charlie”]:
    print name
for i in range(1,10):
    print i
```

Control Structures

- **`break`** statement
  - exits the innermost loop
- **`continue`** statement
  - jump to the beginning of the innermost loop
- Example:

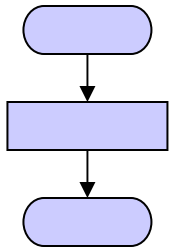
```
i = 0
s = 0
while 1:                # “endless” loop
    i += 1
    if i > 100:
        break;          # exit the loop!
    if i % 2 == 1:
        continue        # next iteration!
    s += i
print s
```

Control Structures

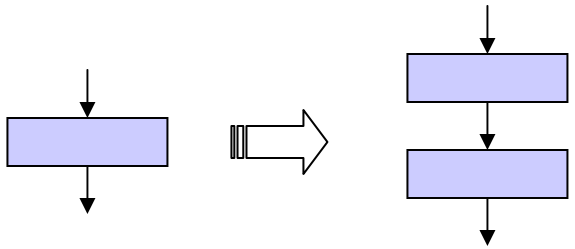
- **`range()`** function
  - returns a list of values in the range specified as argument
  - one argument: **`range(end)`**
    - returns list of values from **`0`** to **`(end-1)`**
    - Example: **`range(5)`** returns **`[0, 1, 2, 3, 4]`**
  - two arguments: **`range(start, end)`**
    - returns list of values from **`start`** to **`(end-1)`**
    - Example: **`range(2, 6`**) returns **`[2, 3, 4, 5]`**
  - three arguments: **`range(start, end, increment)`**
    - returns list of values from **`start`** to **`(end-increment)`** in steps of **`increment`**
    - Example: **`range(6, 2, -1)`** returns **`[6, 5, 4, 3]`**

Structured Programming

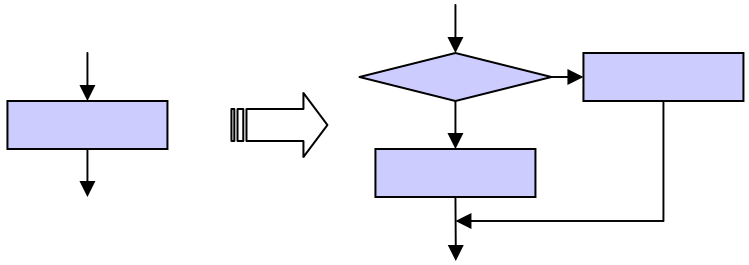
- Initial flow chart
  - Start
  - Program body
  - Finish



- Statement sequences
  - Statement blocks can be concatenated
  - Sequential execution



- Nested control structures
  - control structures can be placed wherever statement blocks can be placed in the code



Example: Average of Numbers

- Version 1: Counter-controlled repetition

(exactly 10 values must be entered)

```
# average.py: compute the average of a set of numbers
#
# author: Rainer Doemer
#
# modifications:
# 01/19/04 RD    initial version

# initialize
count = 0
sum = 0.0

# input and compute
while count < 10:
    x = float(raw_input("Please enter a number: "))
    sum += x
    count += 1

# compute
average = sum / count

# output
print "The sum is", sum
print "The average is", average
```

Example: Average of Numbers

- Version 2: Sentinel-controlled repetition

(number of values entered is determined by the user at run-time)

```
# average2.py: compute the average of a set of numbers
#
# author: Rainer Doemer
#
# modifications:
# 01/19/04 RD       initial version (based on average.py)

# initialize
count = 0
sum = 0.0

# input
s = raw_input("Enter a number or type 'q' to quit: ")

# compute and input
while s != 'q':
    x = float(s)
    sum += x
    count += 1
    s = raw_input("Enter a number or type 'q' to quit: ")

# compute and output
if count > 0:
    average = sum / count
    print count, "numbers entered."
    print "The sum is", sum
    print "The average is", average
else:
    print "No numbers entered."
```

Example: Average of Numbers

- Version 3: **`break`** statement

```
# average3.py: compute the average of a set of numbers
#
# author: Rainer Doemer
#
# modifications:
# 01/19/04 RD       initial version (based on average2.py)

# initialize
count = 0
sum = 0.0

# input and compute
while 1:
    s = raw_input("Enter a number or type 'q' to quit: ")
    if s == 'q':
          break;
    x = float(s)
    sum += x
    count += 1

# compute and output
if count > 0:
    average = sum / count
    print count, "numbers entered."
    print "The sum is", sum
    print "The average is", average
else:
    print "No numbers entered."
```

Functions

- Types
  - Programmer-defined functions
  - Library functions
- Concepts
  - Function call
    - caller invokes a function
  - Function arguments
    - arguments supply data to the function
  - Function parameters
    - input data supplied to the function
  - Return value
    - output data computed by the function
  - Local variables

Functions

- Example
  - y = square(x)

```
# function definition
def square(x):
    y = x * x
    return y

# function call
print square(8.0)
```

  - Function definition
    - function name: square
    - function parameter: x
    - function return value: y
  - Function call
    - argument: 8.0
    - result: 64.0

Example: Compound Interest

- New version with **`interest()`** function

```
# interest2.py: compute compound interest
# author: Rainer Doemer
# modifications:
# 01/26/04 RD  modified for demonstration of functions
# 01/19/04 RD  initial version

# function definition
def interest(principal, rate):
    return principal * (rate/100.0)

# input
amount = float(raw_input("Enter the principal: "))
apr = float(raw_input("Enter the interest rate: "))

# compute and output
for year in range(1,11):
    amount += interest(amount, apr)
    print "End of year %2d: amount on deposit = %8.2f“ \
                % (year,amount)
```

Module **`math`** Functions

- Math module
  - part of Python standard library
  - standard mathematical functions
- Functions provided by **`math`**
  - **`acos()`**
  - **`asin()`**
  - **`atan()`**
  - **`ceil()`**
  - **`cos()`**
  - **`exp()`**
  - **`fabs()`**
  - **`floor()`**
  - **`fmod()`**
  - **`hypot()`**
  - **`log()`**
  - **`log10()`**
  - **`pow()`**
  - **`sin()`**
  - **`sqrt()`**
  - **`tan()`**
  - **`...`**
- Importing functions from the **`math`** module
  - Example
    - **`import math`**
    - **`print math.sqrt(9.0)`**