



THE UNIVERSITY OF
WESTERN
AUSTRALIA

CITS1401

Computational Thinking with Python



Lecture 0

Introduction

My introduction

- Unit Coordinator: Dr. Ghulam **Mubashar** Hassan
- Consultation time : 2 – 3 PM **TUESDAYS**
- My research areas: Artificial Intelligence, Machine Learning
interdisciplinary problems & Engineering Education
- Website: www.csse.uwa.edu.au/~00080148/
- Office: **CSSE Room: 2.12**

Teaching team

- Laboratory demonstrators
 - *Naeha Sharif*
 - *Saqib Ejaz Awan*
 - *Nayyer Aafaq*
 - *Daniel Cowen*
- Admin/enrolments/labs/etc.
 - CSSE Reception (*Rosie Kriskans*) or admin-csse@uwa.edu.au

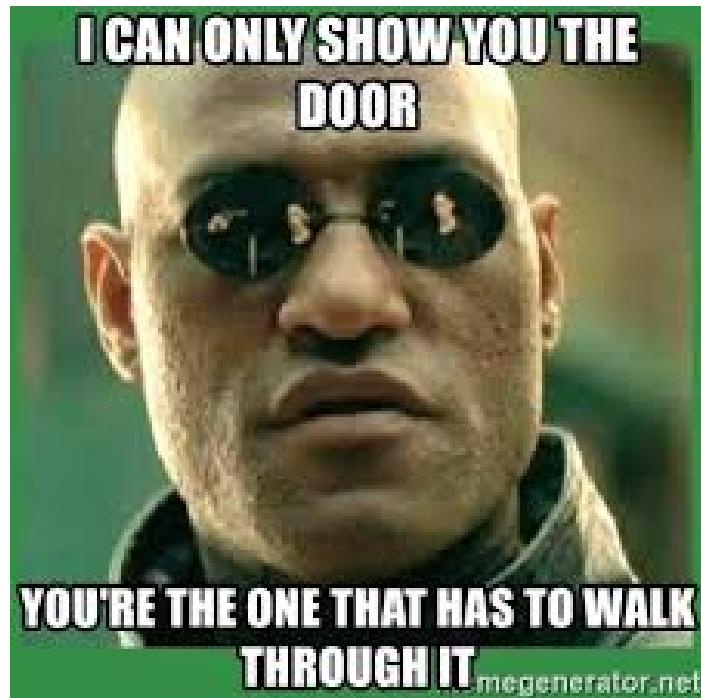
What is CITS1401 About?

- CITS1401 is about computer-based problem solving
 - *How to formulate the problem in a computer language as series of steps*
- Will say a little about computers and how they work, and also about how to solve problems using programs
- Shall be using Python 3 as our computer language
 - *Please do not use Python 2. Related dialect, but incompatible*

Course Outcomes

- Understand how a computer works
- Understand how to express a problem in computational terms
- Be able to write a program in Python 3 to:
 - *Solve small problems*
 - *Automate repetitive computational tasks*

Teaching Strategy



Sooner or later you're going to realize, just as I did, that there is a difference between knowing the path and walking the path.

-Morpheus, The Matrix

TRUSTRUTH.COM

Textbook and Resources

- “Python Programming: An Introduction to Computer Science (3e)”, *John Zelle, Franklin Beedle*.
- All CITS1401 resources (including PDFs of the lectures) can be found on the LMS page for the unit
 - *You need to be enrolled in the unit to see the page*
- All submissions will be made by LMS
- *Few students need to rely on LMS due to travel ban.*

Organisation

- 2 x 1hr lectures a week and 1 x 1hr Workshop
 - *Both lectures and workshop slots will be treated in similar manner. The contents of workshop are embedded in the lectures to make them more interactive.*
 - *They are recorded and available on LCS*
 - *The lectures slides you find on LMS do not necessarily correspond to timetabled lecture/workshop slots.*
 - *Slots are, in fact 45 mins, starting on the hour*

Organisation

- 1 programming lab per week (2 hrs)
 - *Lab demonstrator is available*
 - *Starts Week 2*
- Check your Timetable
 - *multiple time slots across the week*
 - *Feel free to drop in any session if you can find a space*

Labs - Expectations

- Five labs are **assessed** and rest non-assessed
- If you want to do well in the unit you should attend at least one lab session per week
 - *Some learning in the unit, particular related to problem solving, will only take in labs*
- If you cannot finish the lab in time then you can take it home or come to another session. Students are encouraged to work on their own pace.
- You are welcome to attend as many lab sessions as you want
 - *preference to those timetabled to be there*

Labs - Expectations

- You are welcome and encouraged to bring your own laptop with Thonny installed. (You may also bring them in lectures/workshops)
- **This is your time to work on relevant exercises from labsheets with help at hand**
- *The contents covered in labs are part of the course and it may be more than you have covered in the lectures*

Programming Environments

- In the lab you will use Python 3.5 (or above) via the Thonny IDE
 - *An integrated software development environment where you can write, edit, execute and debug programs*
- Thonny is student oriented. It is a free software available for all major operating systems such as Windows, OS, Linux. Python 3.5 or above is built in
 - *Not phones (Android or iOS)*
- You can download Thonny from
<http://www.thonny.org>

Assessment

- Assessment is based on both
 - *Understanding of fundamental concepts*
 - *Practical problem-solving and programming skills*
- Two exams:
 - *Mid-semester Exam : **Wed. in Week 8*** (worth 15%)
 - *Final Exam* (worth 50%)
- Two programming projects
 - *Project 1 due **Mon. 8:00 am of Week 8*** (worth 13%)
 - *Project 2 due **Fri. 5:00 pm of Week 12*** (worth 17%)
- Five lab sheets due ***Fri. 5:00 pm of Week 11*** (worth 5%)

Getting Help

- Discussion Forum on LMS
- Labs
- Textbook
- Above all, seek help early



Svengraph, WikiMedia

Do Something Useful in Week 1

- Get your pHEME login and password
- Organize your UWA email account
- Obtain your timetable (online)
- Get familiar with the CITS1401 LMS website
- Install Thonny (it comes with recent version of Python)

Other Stuff

- Interesting Things page
 - *Feel free to share interesting things. Email them to me*
- Studiosity help link
- I have set slides in Century Schoolbook font (with some Courier and Arial for computer code and meta-language). If you have trouble reading it, please let me know
 - *Accessibility is important*

Other Stuff

- “10 Signs You Will Suck at Programming”
 - *Article linked to Interesting Things page*
 - *Has really great advice about what you need to succeed at programming*
 - *READ IT*
- Engage with the unit!!!
 - *Good data to show that if you turn up to lectures and generally engage with the unit, you will do better (Drouin, 2014, Edwards & Clinton 2018) – see Interesting Stuff*



**KEEP
CALM
AND
HAVE
FUN**

PheobeA

PheobeA - Redbubble

Acknowledgements

- It is important to acknowledge the PPT slides for this unit are based on a slide deck supplied by *John Zelle* (textbook author), though modified, augmented and reordered by *Michael Wise* and *Ghulam Mubashar Hassan*



Lecture 1

Computers and Recipes

Programming

- Why do we need it ?
 - *19.4% of increase since Nov 2013 and projected to grow by 10.21% by May 2023 in Australia [Australian Jobs 2019]*
 - *Everything is getting digitized and we need to interact with computers*
 - *It is problem solving for the most part*
- What is computer like ?
 - *Happy to do whatever asked*
 - *Happy to do repetitive and boring tasks*
 - *Deaf-mute who understands 0's and 1's only*
 - *Having IQ of zero ☺*

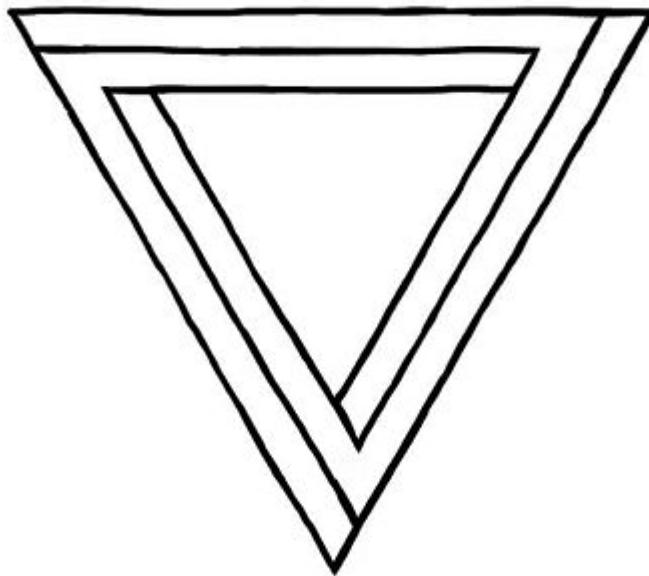
This makes programming a challenging and fun task

Problem Solving Step by Step

- Problem solving starts with breaking down the problem to a series of steps that can be achieved
 - *Problem decomposition*
 - *If these steps still too big, decompose further*
 - *“A journey of a thousand miles begins with a single step” (Chinese saying, based on a quote from Lao Tzu)*

Exercise

- Write step by step instructions for the person sitting next to you to draw the below mentioned image.



Recipe to boil an Egg

- Step 1:

Bring your eggs to room temperature before boiling. If the eggs are too cold, the shells may crack during cooking.

- Step 2:

Place the eggs in a saucepan of cold water. Place the pan over medium heat. Bring to a gentle simmer, gently stirring the eggs constantly in a clockwise direction. The movement of the water helps to centre the egg yolks.

- Step 3:

Simmer the eggs for 4 minutes for soft-boiled eggs. For semi-firm yolks and hard whites, simmer for 5 minutes. For hard-boiled eggs, simmer for 8 minutes. Use a slotted spoon to remove the egg from the water. Transfer to an egg cup and serve immediately.

Source: <http://www.taste.com.au/how+to/articles/2508/how+to+boil+eggs>

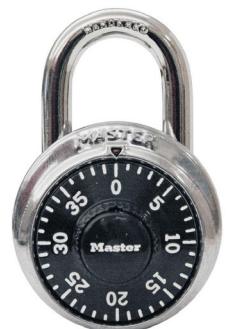
Recipe to boil an Egg (2)

- **Step 1:**
 - *Wait until* your eggs reach room temperature.
- **Step 2:**
 - *Place* the eggs in a saucepan of cold water.
 - *Place* the pan over medium heat.
 - *Until* temperature between 90°C to 95°C *stir* the eggs gently in a clockwise direction. That is a simmer.
- **Step 3:**
 - *If* soft-boiled eggs desired, *simmer* the eggs for 4 minutes
 - else If* semi-firm yolks and hard whites desired, *simmer* for 5 minutes.
 - else if* hard-boiled eggs desired, *simmer* for 8 minutes.
 - Use a slotted spoon to *remove* the egg from the water and *transfer* to an egg cup
 - *Serve* immediately.

Actions words – RED , Control words – BLUE

What is a Computer Program?

- A detailed, step-by-step set of instructions *executed* by a computer
 - *Programming is the creation of the lists of instructions*
- If we change the program, the computer performs a different set of actions or a different task.
- That is, the machine stays the same, but the program changes!
 - *Compare with mechanical systems, e.g. locks.*



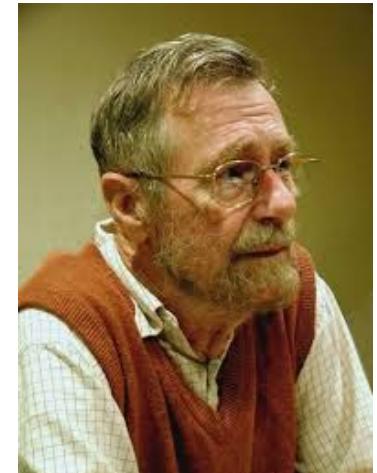
Masterlock.com

What is Computer Science?

- It is NOT the study of computers!

“Computers are to computer science what telescopes are to astronomy.”

Edsger Dijkstra



- Since a computer can carry out any computation, the question really is,

Wikipedia.org

- “*What computations we can describe?*”
- The fundamental question is,
 - “*What can be computed?*”

What is Computer Science?

- Computer scientists find the answers to questions through
 - *Design*
 - *Analysis*
 - *Experimentation*

Design

- One way to show a particular problem can be solved is to actually design a solution.
- This is done by developing an **algorithm**
- *Algorithm*: A step-by-step process for achieving the desired result
 - *An algorithm is simply an abstract recipe*
 - *A program implements that recipe in a particular computer language*
- This Unit will teach you how to
 - *Design an algorithm*
 - *Write a program for it*

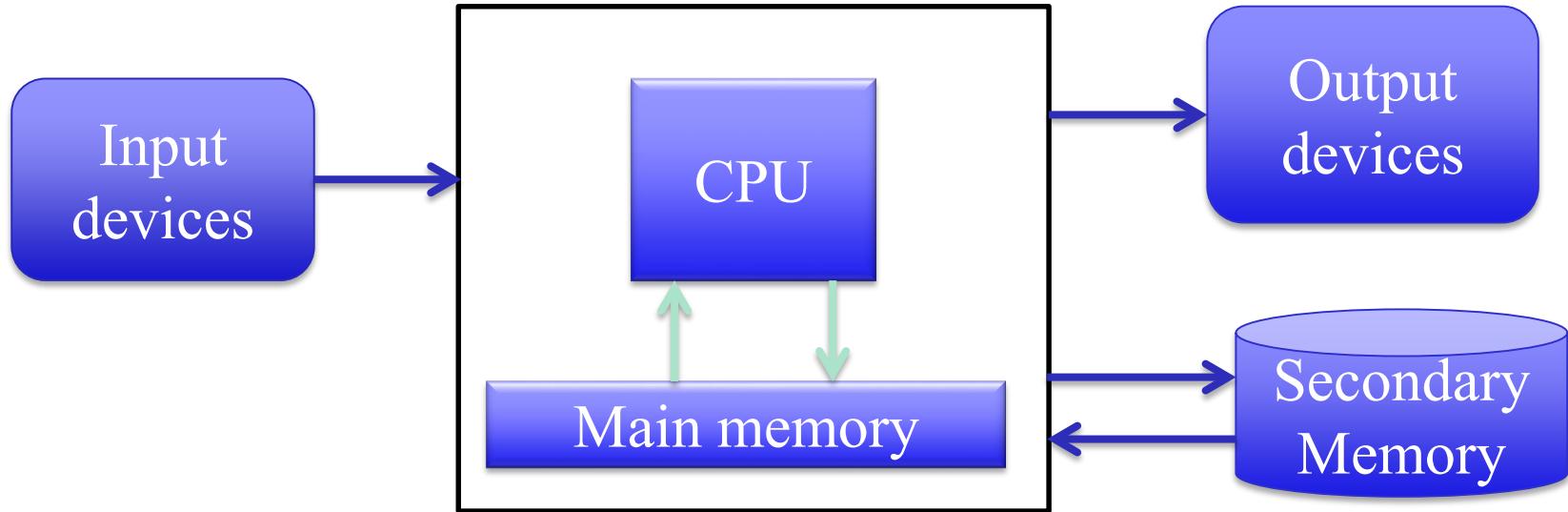
Analysis

- “**Design**” can only answer the question “What is computable?” in the positive.
 - *Not being able to design an algorithm does not mean it is unsolvable.*
- **Analysis** is the process of examining algorithms and problems mathematically.
- Some seemingly simple problems are unsolvable by any algorithm.
 - *Integer partition: Can you partition n integers into two subsets such that the sums of the subsets are equal.*
- Ways of comparing algorithms, e.g. time required to solve problem, or memory required, as a function of the size of the input

Experimentation

- Some problems are too complex for analysis.
 - *World climate*
- Implement a system and then study its behaviour under different conditions
- Experimentation is sometimes still needed after theoretical analysis
 - *To verify the analysis*
 - *To refine the analysis*

Computer Hardware Basics



- The Central Processing Unit (CPU) carries out the computations
 - *Just simple instructions like adding two numbers.*

Computer Hardware

- Memory stores programs and data.
- CPU can only directly access information from the main memory: Random Access Memory (RAM)
- RAM is fast but volatile i.e. all information is lost when power is lost.
- Secondary memory provides more permanent storage (non-volatile).
 - *Magnetic (hard drive)*
 - *Optical (CD, DVD, Blue Ray Disc)*
 - *Solid state drives (USB, SSD memory)*

Input Devices

- Input devices – pass information to the computer
 - *Keyboards and Mice*
 - *Touch pads*
 - *Camera*
 - *Microphone*
 - *Sensors, e.g. accelerometer, gyroscope, data glove*

Output Devices

- Output devices – pass information back to the user or device
 - *Screen*
 - *Printer*
 - *Speaker*
 - *Motor actuator, e.g. robot arm*

The Fetch Execute Cycle

- 
1. Load program into the main memory (RAM)
 2. Fetch the next instruction from memory
 3. Decode the instruction to see what it represents
 - *Fetch data as required*
 4. Carry out the appropriate instruction.

Instructions, data, memory locations – *everything* – is represented as binary numbers

Programming Languages

- Natural languages cannot precisely describe an algorithm.
 - *Try giving directions without waving your arms about your arrival to lecture theatre*
- Programming languages used to express algorithms in a precise way.
- Every structure in a programming language has a precise *form* called its **syntax**.
- Every structure in a programming language has a precise *meaning* called its **semantics**.

Programming Language Levels

- High-level programming languages
 - *Designed to be understood and written by humans*
- Low-level language
 - *Computer hardware can only understand a very low level language known as **machine language***

High-level Programming Language

- In a high-level language, a typical statement may be
 $b = a + 2 \times b$
Note the sequence of operations that is implied
- This needs to be translated to machine language so the computer can execute it
- **Compilers** convert programs written in high-level languages into machine language in one go
- **Interpreters** do the same instruction by instruction

Low-level Language

- The corresponding low-level language may look something like this: if translated in English otherwise it is 0's or 1's

- Load the number from memory location 5001 into CPU Register 0
 - Load the number from memory location 5002 into the CPU Register 1
 - Multiply value in CPU Register 1 by 2 and restore in CPU Register 1
 - Add Register 0 to CPU Register 1 and restore in CPU Register 0
 - Store CPU Register 0 into memory location 5002

Note: A **Register** is a space for temporary results in the CPU (very fast access)

Compiling vs Interpreting

Compiling

- Once program is compiled, the machine language program can be executed over and over without the source code or compiler
- Compiled programs generally run faster since the translation of the program happens only once
- Program needs to be compiled after every minor change in it
- A program compiled for Windows will not run on OS (Mac) or Linux
- C, C++ language programs

Interpreting

- The source code and interpreter are needed each time the program is executed
- Interpreted programs run slower due to each line being interpreted each time it is executed
- More flexible programming environment since programs can be developed and run interactively
- Interpreted programs are more portable across different platforms e.g. Macs, Windows, Linux
- Python, Java language programs

Python 3

- We will be using Python 3 which is embedded in Thonny
- When you start Python, you may see something like:

```
Python 3.6.4
```

```
>>>
```

- `>>>` is a Python 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
>>>
```

Summary

- Understanding the roles of hardware and software in a computing system.
- Learning what computer scientists study and the techniques they use.
- Understanding the basic design of a modern computer.
- Understanding the form and function of programming languages, and how programs in those languages are executed.



THE UNIVERSITY OF
WESTERN
AUSTRALIA

Lecture 2

The Software Development Process

Objectives of This Lecture

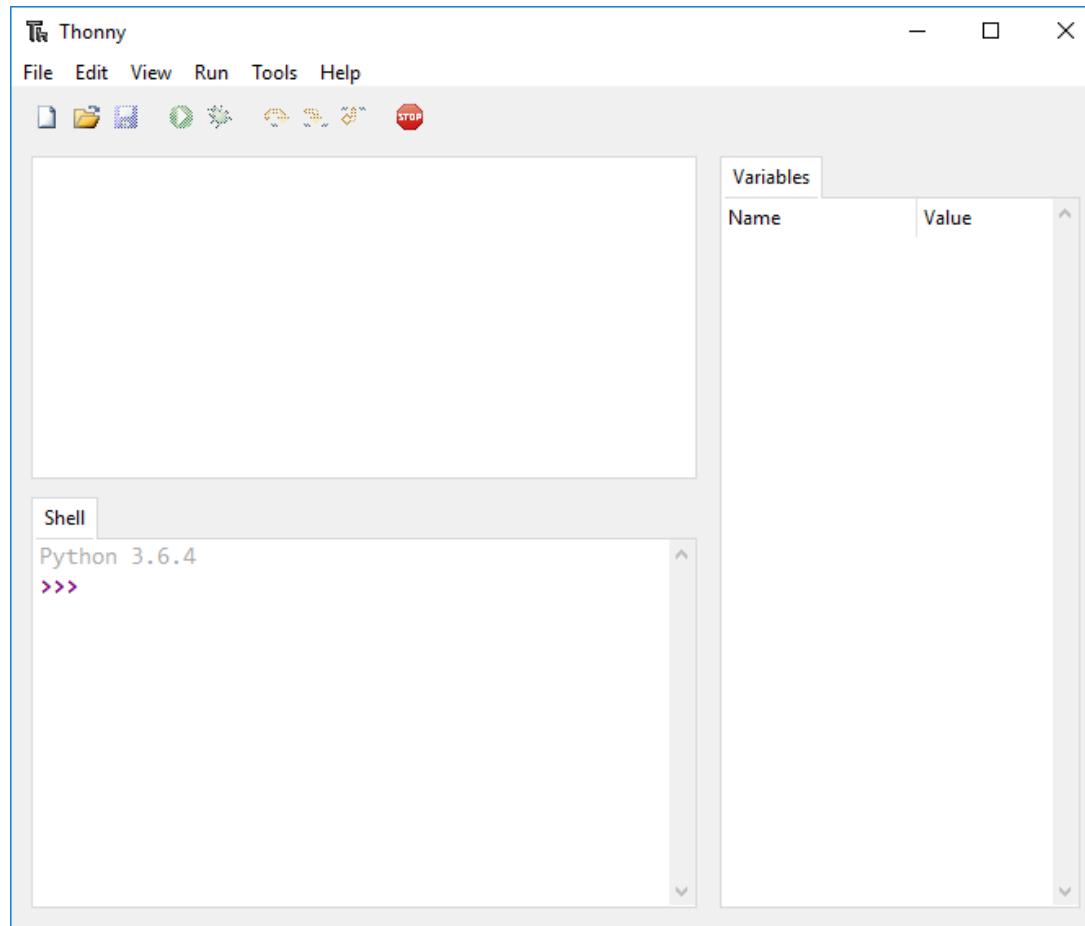
- Getting started with Python/Thonny
- To know the steps of a software development process
- Understand and write simple Python statements
- Understand the concept of pseudocode
- Elements of a program

Getting Started with Python

Start with single statements

```
>>> 2+3  
5  
>>> 22/7  
3.142857142857143  
>>> 3**2  
9  
>>> print("Hello world")  
Hello world  
>>> print("2+3=", 2+3)  
2+3=5
```

Getting stated with Thonny



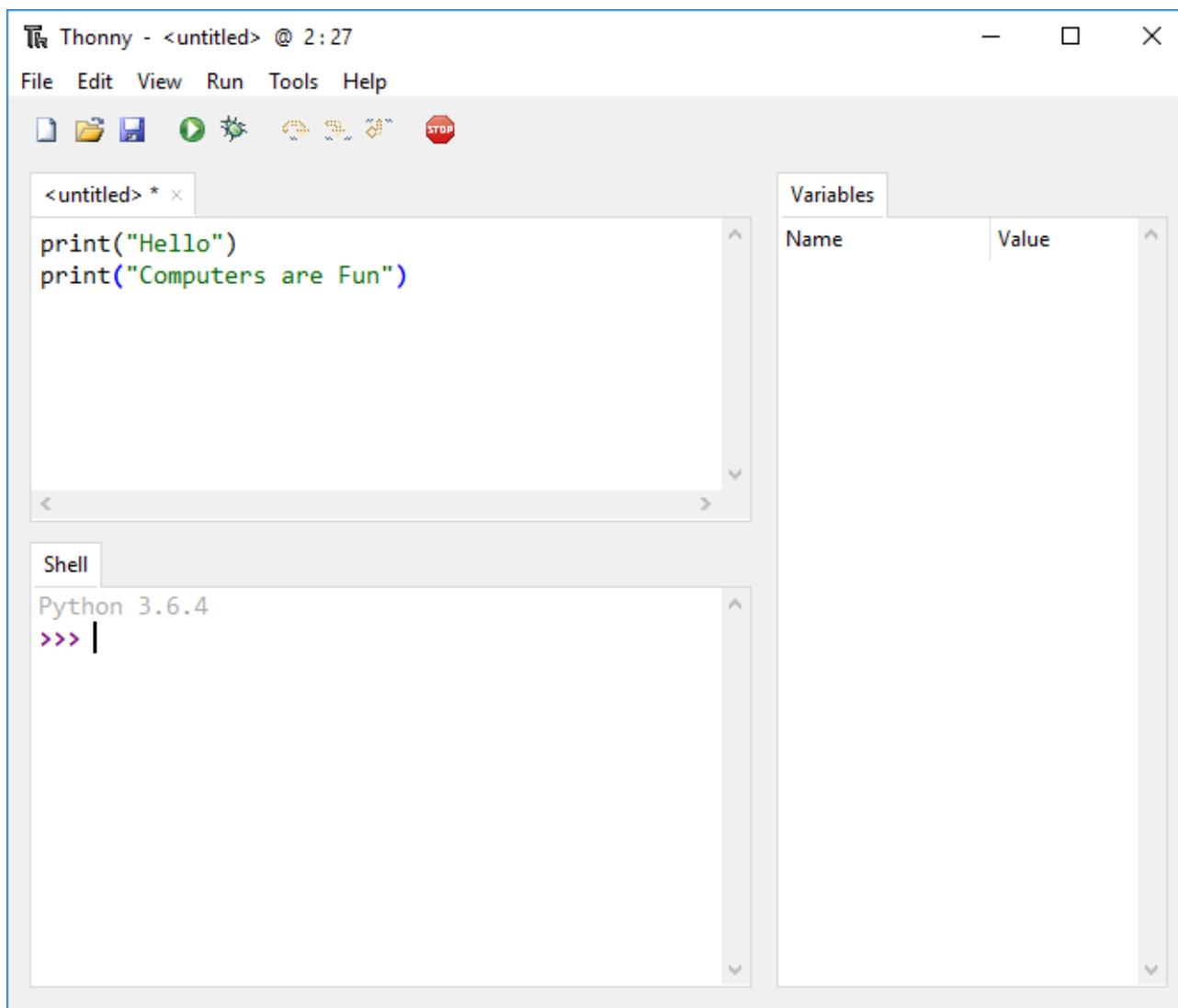
Group Multiple Statements

- To solve a problem, we generally need to execute more than one statements.
- One way to do this is to use a **file**
- Create a file and type the statements

```
print ("Hello")  
print ("Computers are Fun")
```

- Press the green button to run the file
- All statements will be executed line by line

Thonny demo



Analyse the Problem

- Figure out what exactly is the problem to be solved.
- Try to understand it as much as possible.
- You cannot solve a problem unless you fully understand it.
⇒ Talk to users. Better still, *listen* to users

Determine Specifications

- Describe exactly what your program will do
 - *At this stage, don't worry **how** it will do it.*
 - *Only figure out **what** your program will do.*
- Describe the inputs and outputs.
- Describe how the outputs relate to the inputs.

Create a Design

- Formulate the overall structure of the program.
- This is where the “how” of the program gets worked out.
 - *Not code yet.*
- You choose or develop your own algorithm that solves the problem and meets the specifications.

Implement the Design

- Translate the design into a computer language.
- Write each step of the design as program statements.
 - *You will be working individually in this unit, but in industry, teams are involved in projects*
- We will use Python 3 as our programming language.

Test/Debug the Program (Important)

- Your program will often have **syntax errors**.
 - *These are highlighted by the interpreter.*
 - *Need to fix them before the program will work at all.*

```
>>> 32/
```

```
File "<pyshell>", line 1  
32/  
^
```

SyntaxError: invalid syntax

- Even syntactically correct programs may not work as expected.
 - *Logic errors – the sequence of instructions are legal, and the program will run, but do not compute the intended function*
 - *For multiplication of two number 2 & 5*

```
>>> 2**5
```

32

Debugging

- If there are any errors (*bugs*), they need to be located and fixed. This process is called **debugging**.
- **THOROUGH TESTING IS CRUCIAL.**
 - *If you don't find the bugs, the users will !!!*
 - *Your goal is to find errors, so try everything that might "break" your program!*
⇒ **Antibugging** (putting in tests for likely errors)
 - *Try different input values and see if the results are correct.*
 - *Important in industry. More immediately, important for the Projects*



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.
 - *Software Life Cycle*

Example Program: Temperature Converter

- Analysis – the temperature is given in Celsius, user wants it expressed in degrees Fahrenheit.
- Specification
 - *Input – temperature in Celsius*
 - *Output – temperature in Fahrenheit*
 - $Output = 9/5(input) + 32$

Temperature Converter: Design

Design

- *Overall: 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*

Write the Pseudocode First

- Before writing the actual program (code), let's start by writing the **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.
- Difference between algorithm and pseudocode
 - *Algorithms can be described in various ways, from pure mathematical formulas to complex graphs, more times than not, without pseudocode.*
 - *Pseudocode describes how you would implement an algorithm without getting into syntactical details*

Pseudocode

Pseudocode

1. *Prompt the user to input the temperature in degrees Celsius (store it as celsius)*
 2. *Calculate fahrenheit as $(9/5) * celsius + 32$*
 3. *Output fahrenheit*
-
- Now we need to convert this to Python!

Temperature Converter: Python program

```
""" convert.py
```

```
A program to convert Celsius temps to Fahrenheit  
by: Someone Programmer """
```

```
celsius = float(input("What is the Celsius temperature? "))  
fahrenheit = (9/5) * celsius + 32  
print("The temperature is ", fahrenheit, " degrees Fahrenheit.")
```

- Note the multiline comment at the start. It is important as it tells the maintainer:
 - What the program does
 - Statement of authorship

Testing the Program

The next step is to test the program (Press Run or green button on Thonny)

```
>>>  
What is the Celsius temperature? 0  
The temperature is 32.0 degrees Fahrenheit.  
>>>  
What is the Celsius temperature? 100  
The temperature is 212.0 degrees Fahrenheit.  
>>>  
What is the Celsius temperature? -40  
The temperature is -40.0 degrees Fahrenheit.  
>>>
```

Elements of Program: Identifiers

- Names
 - *Names are given to:*
 - variables (e.g. celsius, fahrenheit)
 - functions (e.g. main)
 - modules (e.g. temp_converter, chaos)
 - 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.*

Identifiers examples

- These are all **different**, valid names
 - *X*
 - *Spam*
 - *spam*
 - *spAm*
 - *Spam_and_Eggs*
 - *Spam_And_Eggs*
 - *_X*
 - *C3P0*

Reserved words

- Some identifiers are part of Python itself.
- These identifiers are known as *reserved words*. They are not available for you to use as a name for a variable, etc. in your program.
- `and, def, for, is, raise, assert, elif, in,`
`print, etc.`
- For a complete list, see table 2.1 in the textbook

Elements of Program: Expressions

- Expressions

- *The fragments of code that produce or calculate new data values are called **expressions**.*
$$(9/5) * \text{celsius} + 32$$
- *Expressions are composed of **literals**, variables and operators*
- ***Literals** are used to represent a specific value, e.g. 3.9, -1, 1.0, 3.0e8, "Fred"*
- *Two expressions can be combined with an operator to make another expression*

Expressions

```
>>> x = 5  
  
>>> x      # This only works on interactive interpreter  
5  
  
>>> print(x)  # This works both interactive and from file  
5  
  
>>> print(spam)  
  
Traceback (most recent call last):  
  File "<pyshell#15>", line 1, in -toplevel-  
    print spam  
  
NameError: name 'spam' is not defined  
  
>>>
```

- NameError is the error when you try to use a variable without first having a value having been assigned to it.

Mathematical operators

- Simpler expressions can be combined using *operators*.
- `+, -, *, /, //, **`
- Spaces are irrelevant within an expression
 - *But readability!!*
- The normal mathematical precedence applies.
- $((x1 - x2) / 2*n) + (spam / k**3)$ same as
 $(x1 - x2) / 2*n + spam / k**3$

Elements of Program: Input Information

- The `input` function prints a statement and expects a value (actually a string typed by the user)

```
z = input('type a value ')
```

- The `int` function converts a string of digits to an integer; it will **throw** an **exception** (error) if the user did not type an integer

```
z = int(input('type a value '))
```

- The `float` function works the same way, but expects a floating (decimal) point number

Elements of Program: Output

- Output Statements
 - *A print statement can print any number of expressions (separated by commas).*
 - *Successive print statements will display on separate lines.*
 - *A bare print will print a blank line.*

Print function

Expression	Produces
print(3+4)	7
print(3, 4, 3+4)	3 4 7
print()	
print(3 + 4)	7
print("The answer is", 3+4)	The answer is 7

Lecture Summary

- We learned about the steps of a software development process
- We wrote and analysed simple Python statements
- We learned the concept of pseudocode
- We learned about the importance of testing
- We learned about the elements of a program



Lecture 3

How to write code in Python

Revision: Getting Started with Python

Start with single statements

```
>>> 2+3  
5  
>>> 22/7  
3.142857142857143  
>>> 3**2  
9  
>>> print("Hello world")  
Hello world  
>>> print("2+3=", 2+3)  
2+3=5
```

Functions Group Multiple Statements

- To solve a problem, we generally need to execute more than one statements.
- One way to do this is to use a *file*
- Another way to do this is to use a **function**

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

```
>>>
```

Defining Functions in Python

```
>>> 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. **Indent must be uniform**
- The blank line (hit enter/return twice) on shell lets Python know the definition is finished.

Executing, or Invoking, a Function

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

```
>>>
```

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

```
>>> hello() ← Brackets are important!
Hello
Computers are Fun
>>>
```

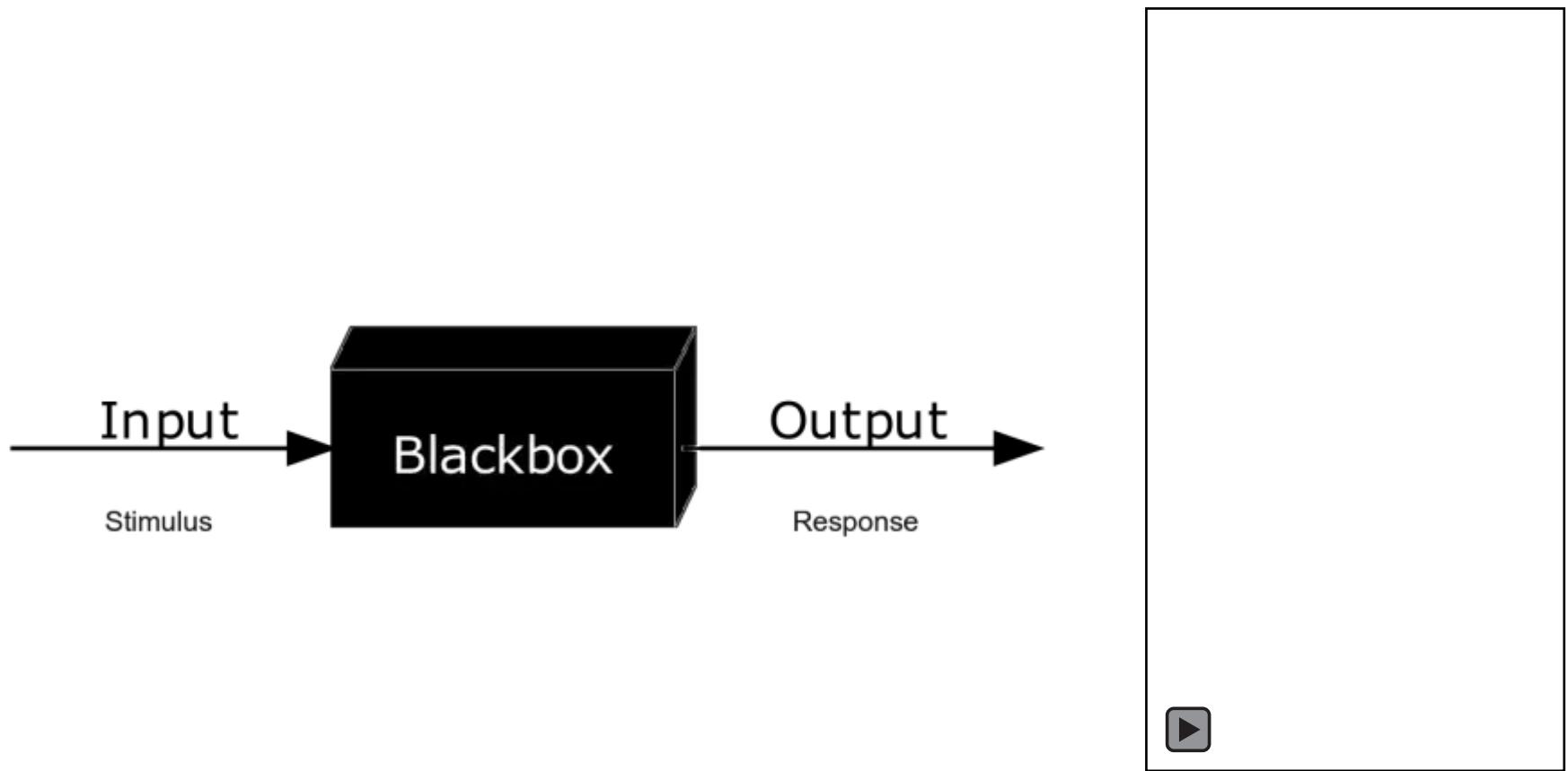
Problems with scripts (files without functions)

- In the scripts we have seen, the entire script consists of just one “block” of statements, executed in order – but this can become unmanageable when your script is thousands of lines long, contained in just one file.
- You may want to use previously written code in other programs. But cutting and pasting bits of code can create inadvertent variable name clashes.
- Scripts are not as flexible as we would like.
- How to address this?

A general problem solving technique is to break down complex problems into smaller, more manageable tasks.

Functions help us to avoid those problems

Functions as black box



Functions can take Inputs (Parameters)

- Functions can have changeable parts called parameters that are placed between the brackets.
- The function “hello” did not need any parameters.
- Here is another function that has one parameter.

```
>>> def greet(person) :  
        print("Hello", person)  
        print ("How are you?")
```

```
>>>
```

Invoking a Function that has parameter(s)

- A function that has **parameters** requires **arguments**
- If we try to execute the function `greet ()`

```
>>> greet()  
Traceback (most recent call last):  
  File "<pyshell#74>", line 1, in <module>  
    greet()  
TypeError: greet() takes exactly 1 argument (0 given)
```

- It gives us an error because we did not specify a value for the parameter “person”

Passing Parameters to Functions

```
>>> greet("Terry")
Hello Terry
How are you?
>>> greet("Paula")
Hello Paula
How are you?
>>>
```

- When we use parameters, we can customize the output of a function.

Why You need to Use Functions

- Define once, use many times
 - *Replace repeated code sections with a parameterized function*
- Aids problem decomposition
 - *Even if code is used just once, helps break problem into smaller, manageable pieces*
 - *Like sections and paragraphs in a paper, or chapters and paragraphs in a story*
- Defining code as functions allows independent testing/validation of code

Functions in a file

- When we exit the Python interpreter, all functions that we defined will cease to exist.
- How about writing them in a file and saving it.
 - *Saves a LOT of retyping*
- A *programming environment* is designed to help programmers write programs and usually include automatic indenting, highlighting, etc.

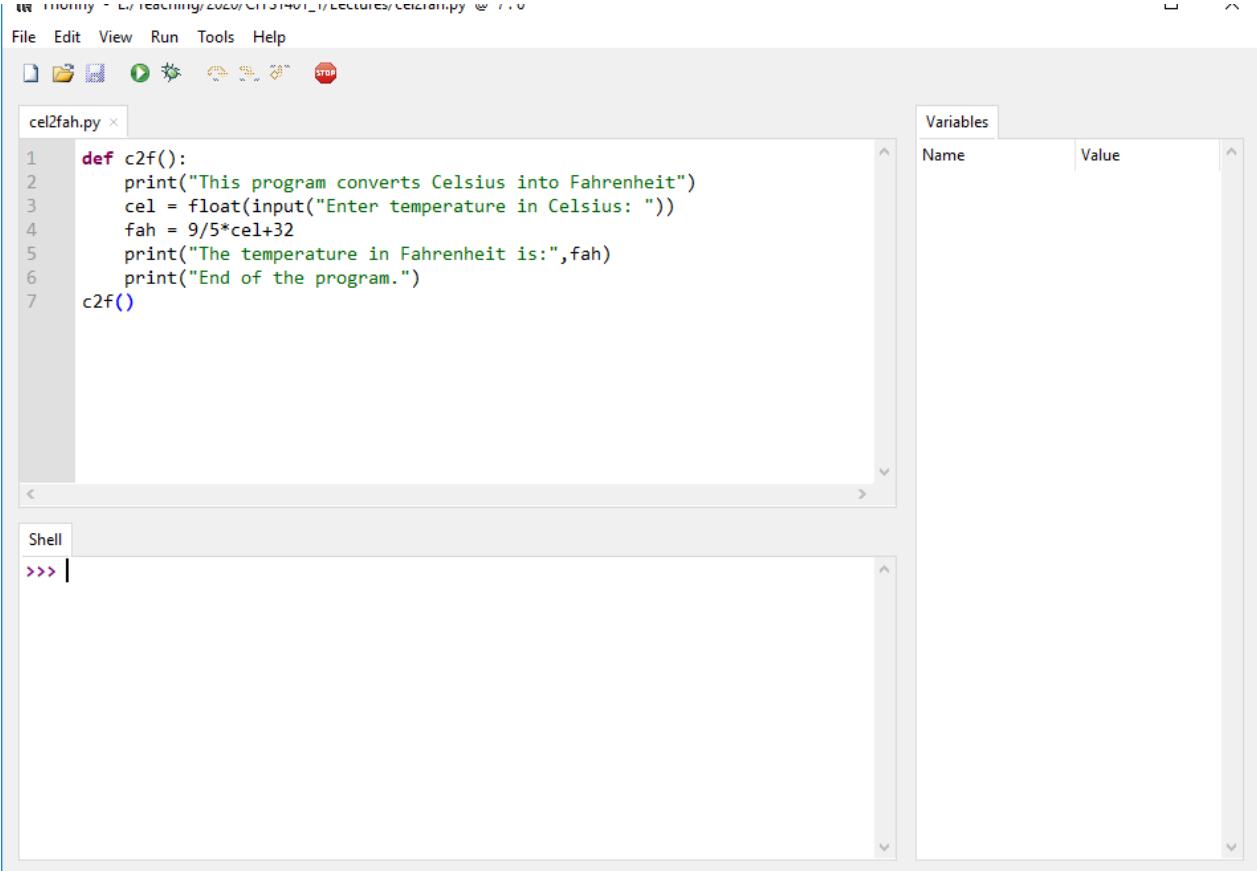
Creating a Module File

```
# File: cel2fah.py
# A simple program is illustrating Celsius to Fahrenheit conversion

def c2f():
    print("This program converts Celsius into Fahrenheit")
    cel = float(input("Enter temperature in Celsius: "))
    fah = 9/5*cel+32
    print("The temperature in Fahrenheit is:", fah)
    print("End of the program.")
c2f()
```

- We use a filename ending in .py when we save our work to indicate it's a Python program.
- Click green button (run) on Thonny to run the program.

cel2fah.py using Thonny IDE



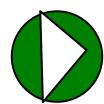
The screenshot shows the Thonny IDE interface. The top menu bar includes File, Edit, View, Run, Tools, and Help. Below the menu is a toolbar with icons for file operations, run, stop, and other tools. The main area contains a code editor titled "cel2fah.py" with the following Python script:

```
def c2f():
    print("This program converts Celsius into Fahrenheit")
    cel = float(input("Enter temperature in Celsius: "))
    fah = 9/5*cel+32
    print("The temperature in Fahrenheit is:",fah)
    print("End of the program.")
c2f()
```

To the right of the code editor is a "Variables" table with one row:

Name	Value

At the bottom left is a "Shell" window with the prompt ">>> |".



Is the **Run** button, or choose from main menu

Running cel2fah.py using Thonny

The screenshot shows the Thonny Python IDE interface. The top menu bar includes File, Edit, View, Run, Tools, and Help. Below the menu is a toolbar with icons for file operations and run/stop. The main area contains a code editor with the file 'cel2fah.py' open, displaying the following code:

```
1 def c2f():
2     print("This program converts Celsius into Fahrenheit")
3     cel = float(input("Enter temperature in Celsius: "))
4     fah = 9/5*cel+32
5     print("The temperature in Fahrenheit is:",fah)
6     print("End of the program.")
7 c2f()
```

To the right of the code editor is a 'Variables' window showing a single entry:

Name	Value
c2f	<function c2f at 0x10303030303030303>

At the bottom left is a 'Shell' window showing the output of running the script and executing the function directly:

```
>>> %Run cel2fah.py
This program converts Celsius into Fahrenheit
Enter temperature in Celsius: 0
The temperature in Fahrenheit is: 32.0
End of the program.

>>> c2f()
This program converts Celsius into Fahrenheit
Enter temperature in Celsius: 40
The temperature in Fahrenheit is: 104.0
End of the program.

>>> |
```

Inside a Python Program

```
# File: cel2fah.py
# A simple program is illustrating Celsius to Fahrenheit conversion
```

- Lines that start with `#` are called *comments*. Similar to text enclosed in triple quotes as discussed in earlier lecture
 - *Comments can begin in the middle of lines, too*
- Intended for human readers and ignored by Python
 - **Important**, so you or other maintainers of that code know what you were intending
 - *Helps maintainability*
- Python skips text from `#` to end of line

Inside a Python Program

```
def c2f():
```

- Beginning of the definition of a function called *c2f*
 - *Note the :* is important. It separates header from the function body

Inside a Python Program

```
print(" This program converts Celsius into Fahrenheit")
```

- This line causes Python to print a message introducing the program to the user.
 - *The message is sent to **Standard Output** (usually the computer screen)*
 - ***Standard Input** is usually the keyboard*

Inside a Python Program

```
cel = float(input("Enter temperature in Celsius: "))
```

- cel is an example of a *variable*
 - A variable is used to assign a name to a memory location so that a value can be stored there and later retrieved.
 - Variables come into existence when first assigned to
 - The quoted text is displayed. The user enters a number (which is text, i.e. just numerical letters).
 - The function float converts the string, e.g. “0.5”, into the number 0.5, which is then stored in cel.
 - Note function call within function call (inner one called first)
-

Inside a Python Program

```
fah = 9/5 * cel + 32
```

- This is called an *assignment* statement
- The part on the right-hand side (RHS) of the = is a **mathematical expression**
- *, + and / are used to indicate multiplication, addition and division respectively
- Once the value on the RHS is computed, it is stored back into (*assigned*) into fah

```
print("The temperature in Fahrenheit is:", fah)
```

- Prints the calculated temperature fah to standard output

Indenting your Python programs

```
# File: cel2fah.py
# A simple program is illustrating Celsius to Fahrenheit conversion

def c2f():  
    ←print("This program converts Celsius into Fahrenheit")  
    ←cel = float(input("Enter temperature in Celsius: "))  
    ←fah = 9/5*cel+32  
    ←print("The temperature in Fahrenheit is:", fah)  
    ←print("End of the program.")
```

- Indentation is used in Python programs to indicate the different **blocks** of statements. These are executed together, one after the other
- Note the colon highlighted in purple

Inside a Python Program

```
c2f()
```

- The interpreter first creates a function definition
- The last line tells Python to *execute* the code in the function `c2f`
 - *No arguments expected so none supplied*

Executing a Python Program from a File

- You can run a program in a file any time you want using one of the following methods:
 1. Using Thonny, the easiest way is to click the green forward arrow or select **Run** from the **Run Module**
 2. On the command line (windows) or terminal (Mac OS), enter `python ./cel2fah.py` (`./` generally not be need if **path variable** has been specified)
 - *Paths are where system looks for files and programs*
 3. You can also double click the `.py` file in Windows to run it

Importing a module

```
>>> import cel2fah ← Note: No .py suffix!
```

This program converts Celsius into Fahrenheit

Enter temperature in Celsius: 0

The temperature in Fahrenheit is: 32.0

End of the program.

```
>>>
```

- This tells Python interpreter to load the file cel2fah.py into the main memory.
- Since the last statement of cel2fah.py is c2f() the function will get executed upon importing the file.
- Importing modules very common (particularly library modules – huge range, performing many useful functions)

Importing a Module

- When Python imports a module, it executes each line.
 - *The def causes Python to create the function c2f:*
 - *c2f () call at the end executes the function*
- Upon first import, Python creates a companion file with .pyc extension. This is an intermediate file containing the **byte code** used by the interpreter.
- Modules need to be imported in a session only once.

Modules and Functions

- You can define multiple functions in a module file
- You can call a function by typing
moduleFileName.functionName(...)
 - E.g. `>>> cel2fah.c2f()`
`>>> math.sqrt(2)`

A Word about Eval – Eval is Evil

- The second edition of Zelle uses the eval function:

```
x = eval(input("Enter a number between 0 and 1: "))
```
- This is very risky and no longer considered best practice
- eval evaluates **whatever** you give it, so long as the data is syntactically legal.
- Someone who knows Python could exploit this ability and enter malicious instructions, e.g. capture private information or delete files on the computer.
- This is called a code injection attack, because an attacker is injecting malicious code into the running program.
- When the input is coming from untrusted sources, like users on the Internet, the use of eval could be disastrous.

Summary

- Python is an interpreted language. We can execute commands directly in a shell or write a Python file.
- A Python program is a sequence of commands (statements) for the interpreter to execute. It can take input from the user, print output to the screen and run a set of statements.



THE UNIVERSITY OF
WESTERN
AUSTRALIA

Lecture 4

Variables and simple loops

Revision: functions and module file

```
# File: cel2fah.py
# A simple program is illustrating Celsius to Fahrenheit conversion

def c2f():
    print("This program converts Celsius into Fahrenheit")
    cel = float(input("Enter temperature in Celsius: "))
    fah = 9/5*cel+32
    print("The temperature in Fahrenheit is:", fah)
    print("End of the program.")
c2f()
```

- We use a filename ending in .py when we save our work to indicate it's a Python program.
- Click green button (run) on Thonny to run the program.

Objectives of this Lecture

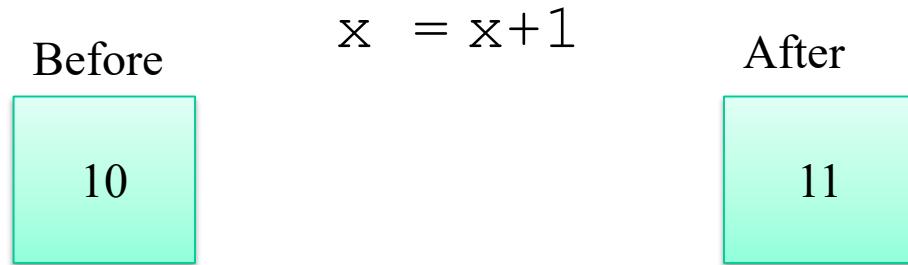
- To understand the process of assigning values to variables
- To look into limitations of data types
- To understand simple loops
- Follow the software development process to develop a program that calculates the future value of investment

Assignment Statements

- Simple Assignment
- **<variable> = <expr>**
variable is an identifier, **expr** is an expression
- The expression on the right is evaluated to produce a value which is then associated with the variable named on the left.

Assignment Statements – the Simple View

- 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.



Assignment Statements

- `x = 3.9 * x * (1-x)`
- `fahrenheit = 9/5 * celsius + 32`
- `x = 5`

The spacing around parts of assignments is optional, but makes the resulting program much more readable – and hence maintainable.

Assignment Statements

- Variables can be reassigned as many times as you want!

```
>>> myVar = 0
>>> myVar
0
>>> myVar = 7
>>> myVar
7
>>> myVar = myVar + 1
>>> myVar
8
```

Simultaneous Assignment

- Several values can be calculated at the same time
- $\langle \text{var} \rangle, \langle \text{var} \rangle, \dots = \langle \text{expr} \rangle, \langle \text{expr} \rangle, \dots$
- Evaluate the expressions on the right and assign them to the variables on the left
 - *Must have same number of expressions as variables!*

```
>>>sum, diff = x+y, x-y
```

Simultaneous Assignment

- How could you swap the values for x and y?

- *Why doesn't this work?*

$x = y$

$y = x$

- *#assume x= 1, y= 9 initially..... x=y will make x= 9 (original x is lost)*

- We could use a temporary variable...

$temp = x$

$x = y$

$y = temp$

Simultaneous Assignment

We can swap the values of two variables easily in Python!

```
>>> x = 3  
>>> y = 4  
>>> print(x, y)  
3 4  
>>> x, y = y, x  
>>> print(x, y)  
4 3
```

Increment Assignment

- Certain types of assignment statement are so common that a short-cut exists

$x = x + n$, (especially $x = x + 1$)

$x = x - n$ (n can be any expression)

- These become

$x += n$

$x -= n$

- Also

$x *= n$

$x /= n$, etc

Numeric Data Types

- There are two different kinds of numbers!
 - *5, 4, 3, 6 are whole numbers – they don't have a fractional part*
 - *0.25, 1.10, 3.142 are decimal fractions*
- 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/variable determines
 - *what values it can have*
 - *and what operations can be performed on it*
 - *Taking 3 from 10: easy*
 - *Taking 3 from J: ?*

Numeric Data Types

- Whole numbers are represented using the *integer* (*int* for short) data type.
 - *Size depends on machine using it*
- *long* is another data type similar to *int* but can store longer number
 - *Needs more memory space*
- These values can be positive or negative whole numbers.

Numeric Data Types

- Numbers that can have fractional parts are represented as *floating point* (or *float*) values.
- How can we tell which is which?
 - *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)*

Numeric Data Types

- Why do we need two number types?
 - *Values that represent counts can't be fractional*
 - *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!*

Numeric Data Types

- Operations on ints produce ints (excluding /)
- Operations on floats produce floats.

```
>>> 3.0+4.0
```

```
7.0
```

```
>>> 3+4
```

```
7
```

```
>>> 10.0/3.0
```

/ does floating point division

```
3.3333333333333335
```

```
>>> 10/3
```

```
3.3333333333333335
```

```
>>> 10 // 3
```

// does integer division

```
3
```

Numeric Data Types

- Integer division produces a whole number.
 - *That's why* $10 // 3 == 3$
- Think of it as ‘**goes into**’, where $10//3 == 3$ since 3 (goes into) 10, three times (with a remainder of 1)
- $10 \% 3 = 1$ is the remainder of the integer division of 10 by 3.

Limits of Int

- What's going on?
 - *While there are an infinite number of integers, there is a finite range of integers that can be represented by int.*
 - *This range depends on the number of bits a particular CPU uses to represent an integer value.*

Limits of Int

- Typical PCs use 64 bit integers.
 - That means there are 2^{64} possible values, centered at 0.
 - This range is -2^{63} to $2^{63}-1$. We need to subtract one from the top end to account for 0.
 - Python solution (recent Python versions): expanding int
-
- Does switching to *float* data types get us around the limitations of *int*?

```
>>> x1 = 10**121
10000...
>>> y1 = x1 + 1
10000... ... ... 1
>>> z1 = x1 - y1
-1
>>> x2 = 10e121
1e+122
>>> y2 = x2 + 1
1e+122
>>> z2 = x2 - y2
0.0
```

Handling Large Integers

- Floats are approximations
- Floats allow us to represent a larger range of values, but with fixed precision.
- Python int is not a fixed size, but expands to handle whatever value it holds.
- Newer versions of Python automatically convert int to an expanded form when it grows so large as to overflow.
- Can store and work with indefinitely large values (e.g. 100!) at the cost of speed and memory

Type Conversion

- Combining an `int` with a `float` in an expression will return a `float`
- We can explicitly convert between different data types
- For example the `int` and `round` functions convert a `float` to integer

```
>>> int(6.8)      # Truncate
```

6

```
>>> round(6.8)
```

7

```
>>> float(6)
```

6.0

Type Conversion : More examples

Conversion function	Example use	Value returned
int(<a number or string>)	int(2.87)	2
int	int("55")	55
float(<a number or string>)	float(32)	32.0
str(<any value>)	str(43)	'43'

str: string/text

Find Data Type

- We can use the **type** function to find the data type

```
>>> type(4)  
<class 'int'>
```

```
>>> type(4.3)  
<class 'float'>
```

```
>>> x = 5.76  
>>> type(x)  
<class 'float'>
```

```
>>> type(hello) # Without quotes, assumes 'hello' is a  
# variable. Not defined, so will generate  
# an error  
>>> type('hello')  
<class 'str'>
```

If a variable exists, the type of the variable is the type of the value assigned to it

Scientific Notation

Decimal Notation	Scientific Notation	Meaning
2.75	2.75e0	2.75×10^0
27.5	2.75e1	2.75×10^1
2750.0	2.75e3	2.75×10^3
0.0275	2.75e-2	2.75×10^{-2}

- Python floating point values go from -10^{308} to 10^{308}
- Typical precision is 16 digits (decimal places)

Float Problems

- Very large and very small floating point values can also cause problems (current Python)

```
>>> x = 1.0e308  
>>> x  
1e+308  
>>> x = 100*x  
>>> x  
inf
```

This is called
over-flow

```
>>> x = 1.0e-323  
>>> x  
1e-323  
>>> x = x / 100.0  
>>> x  
0.0
```

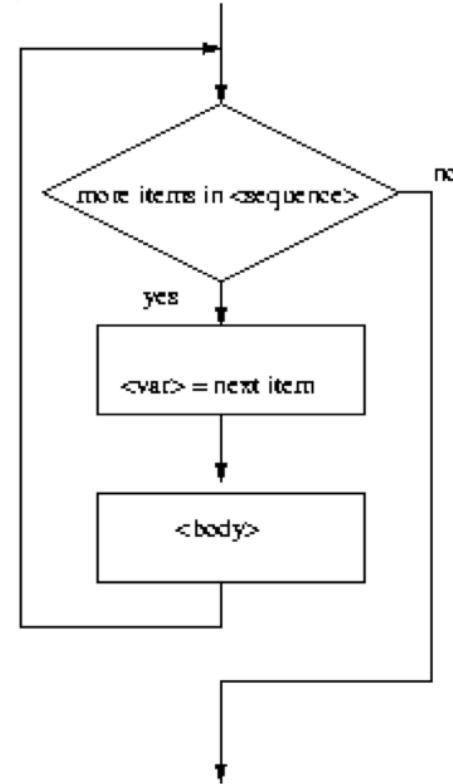
This is called
under-flow

Simple loops



Definite Loops

for loops alter the flow of program execution, so they are referred to as **control structures**.



Definite Loops

- A definite loop executes a pre-specified number of times, **iterations**, which is known when program loaded
- `for <var> in <sequence>:`
`<body>`
- The beginning and end of the body are indicated by indentation.
- Note that iterations are over sequences (more of this in a later lecture)
 - *For the time being, a **sequence** is a countable sequence of Python things (objects)*

Definite Loops

- `for <var> in <sequence>:`
`<body>`
- The variable after the `for` is called the **loop index**. It takes on each successive value in sequence

Definite Loops

- In `chaos.py` (from Labsheet 00), what did `range(10)` do?

```
>>> list(range(10))  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- `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
- The body of the loop executes 10 times

Definite Loops

```
>>> for odd in [1, 3, 5, 7]:  
    print(odd*odd)  
  
1  
9  
25  
49
```

Loop variable `odd` first has the value 1, then 3, then 5 and finally 7

Example Program: Interest Earned

Analysis

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

Example Program: Future Value

Specification

- **Inputs**

principal The amount of money being invested, in dollars

apr The *annual percentage rate* expressed as a **floating point** decimal number $0.0 < \text{apr} < 1.0$

- **Output**

The value of the investment 10 years in the future

- **Relationship**

Value after one year is given by $\text{principal} * (1 + \text{apr})$.

This needs to be done 10 times.

Example Program: Future Value

Design

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

Example Program: Future Value

Implementation

- *Each line translates to one line of Python (in this case)*

- *Print an introduction*

```
print("This program calculates the future")
print("value of a 10-year investment.")
```

- *Input the amount of the principal*

```
principal = float(input("Enter the initial principal: "))
```

Example Program: Future Value

- *Input the annual percentage rate*

```
apr = float(input("Enter the annual interest rate: "))
```

- *Repeat 10 times:*

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

- *Calculate $principal = principal * (1 + apr)$*

```
principal *= (1 + apr)
```

- *Output the value of the principal at the end of 10 years*

```
print("The value in 10 years is:", principal)
```

Example Program: futval.py

```
#     A program to compute the value of an investment
#     carried 10 years into the future
#     Author: Unit Coordinator

def main():
    print("This program calculates the future")
    print("value of a 10-year investment.")

    principal = float(input("Enter the initial principal: "))
    apr = float(input("Enter the annual interest rate: "))

    for i in range(10):
        principal *= (1 + apr)

    print ("The value in 10 years is:", principal)

main()
```

Example Program: Testing futval.py

```
>>> main()
```

This program calculates the future value of a 10-year investment.

```
Enter the initial principal: 100
```

```
Enter the annual interest rate: 0.03
```

```
The value in 10 years is: 134.391637934
```

```
>>> main()
```

This program calculates the future value of a 10-year investment.

```
Enter the initial principal: 100
```

```
Enter the annual interest rate: 0.10
```

```
The value in 10 years is: 259.37424601
```

Lecture Summary

- Understanding the concept of assignment in Python
- We learned how to
 - *assign values to variables*
 - *do multiple assignments in one statement*
 - *definite simple definite loops*
 - *Limitations of data types*
- “**for**” loop alters the sequence of the program



Lecture 5

Math module

Objectives of this Lecture

- A little revision
- To learn how to use the Python math library
- Example: Quadratic equation
- To understand the accumulator program
- Example: Factorial

REVISION: The Software Development Process

- 1. Analyse the Problem
- 2. Determine the Specifications
- 3. Create a Design
- 4. Implement the Design i.e. write the program
- 5. Test/Debug the Program
- 6. Maintain the Program

Software Life Cycle

REVISION: Numerical data types

Conversion function	Example use	Value returned
int(<a number or string>)	int(2.87)	2
int	int("55")	55
float(<a number or string>)	float(32)	32.0
str(<any value>)	str(43)	'43'

What happens if you type: int ("2.87") ?

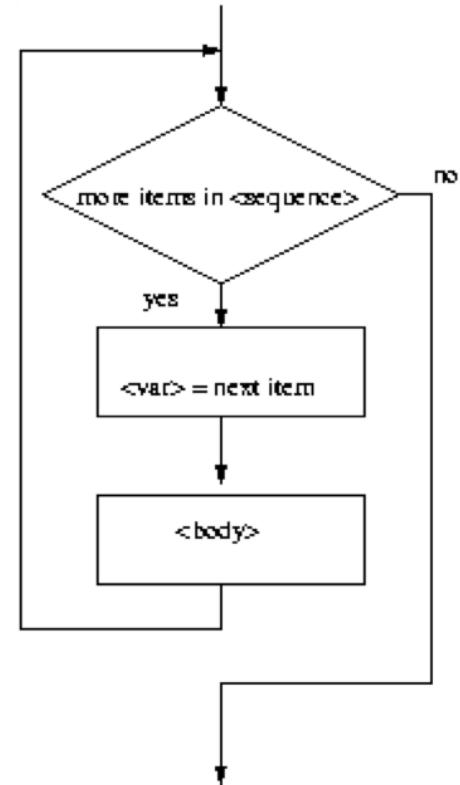
REVISION: Definite Loops

for loops alter the flow of program execution, so they are referred to as **control structures**.

```
>>> for odd in [1, 3, 5, 7]:  
    print(odd*odd)
```

1
9
25
49

Loop variable odd first has the value 1, then 3, then 5 and finally 7



REVISION: module and function

- Module:
 - allows you to logically organize your Python code
 - grouping related code that makes the code easier to understand and use
 - Simply, a file consisting of Python code which can define functions, classes, variables and may also include runnable code
- Function:
 - a block of organized, reusable code that is used to perform a single, related action.
 - provide better modularity for your application and a high degree of code reusing

Using the Math Library

- Besides `(**`, `*`, `/`, `%`, `//`, `+`, `-`, `abs`), we have lots of other math functions available in a *math library*.
 - `**` is *exponentiation*, e.g. `2 ** 3 == 8`
- A *library* is a **module** with some useful definitions/functions.

Using the Math Library

- The formula for computing the roots of a quadratic equation of the form: $ax^2 + bx + c$ is

$$root = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- The only part of this that we don't know how to do yet is the square root function, but that 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      # not mathS
```

- Importing a library makes whatever functions are defined within it available to the program (from that point onwards).

Using the Math Library

- 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, you can do
`discRoot = math.sqrt(b*b - 4*a*c)`

Using the Math Library

- We can also import math module as

```
import math as m # now math is called by m  
m.sqrt(9)
```

- If we do not want to import all functions of module

```
# Import sqrt function only  
from math import sqrt  
sqrt(9) # No need a reference
```

Using the Math Library

```
# quadratic.py
#     A program that computes the real roots of a quadratic equation.
#     Illustrates use of the math library.
#     Author: Unit Coordinator

import math # Makes the math library available.

def main():
    print("This program finds the real solutions to a quadratic equation")
    print()

    a = float(input("Please enter coefficient a: "))
    b = float(input("Please enter coefficient b: "))
    c = float(input("Please enter coefficient c: "))

    discRoot = math.sqrt(b * b - 4 * a * c)
    root1 = (-b + discRoot) / (2 * a)
    root2 = (-b - discRoot) / (2 * a)

    print()
    print("The solutions are:", root1, root2 )
    return
```

Using the Math Library

Running the program

```
>>> main()
This program finds the real solutions to a quadratic
Please enter coefficient a: 3
Please enter coefficient b: 4
Please enter coefficient c: -1
```

The solutions are: 0.215250437022 -1.54858377035

Using the Math Library

Try again with 1, 2, 3

```
>>> main()
This program finds the real solutions to a quadratic
Please enter coefficient a: 1
Please enter coefficient b: 2
Please enter coefficient c: 3

Traceback (most recent call last):
  File "quadratic_roots.py", line 21, in <module> main()
    File "quadratic_roots.py", line 15, in main
      discRoot = math.sqrt(b * b - 4 * a * c)
ValueError: math domain error
```

>>>

Runtime error

Math Library

- If $a = 1$, $b = 2$, $c = 3$, then we are trying to take the square root of a negative number!
- Using the `sqrt` function is more efficient than using `**`
 - *How could you use `**` to calculate a square root?*

Math Library

Python	Mathematics	English
pi		An approximation of pi
e	e	An approximation of e
sqrt(x)		The square root of x
sin(x)	$\sin x$	The sine of x
cos(x)	$\cos x$	The cosine of x
tan(x)	$\tan x$	The tangent of x
asin(x)	$\arcsin x$	The inverse of sine x
acos(x)	$\arccos x$	The inverse of cosine x
atan(x)	$\arctan x$	The inverse of tangent x

Don't forget to put `math` in front (depending on the method of importing module, e.g. `math.sqrt(x)`, `math.pi`)

Math Library

Python	Mathematics	English
<code>log(x)</code>	$\ln x$	The natural (base e) logarithm of x
<code>log10(x)</code>	$\log_{10} x$	The common (base 10) logarithm of x
<code>exp(x)</code>	e^x	The exponential of x
<code>ceil(x)</code>	$\lceil x \rceil$	The smallest whole number $\geq x$
<code>floor(x)</code>	$\lfloor x \rfloor$	The largest whole number $\leq x$

Accumulating Results: Factorial

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

Accumulating Results: Factorial

- How we could write a program to do this?
- Input number to take factorial of, n
Compute factorial of n, fact
Output fact

Accumulating Results: Factorial

- How did we calculate $6!$?
- $6 * 5 = 30$
- Take that 30, and $30 * 4 = 120$
- Take that 120, and $120 * 3 = 360$
- Take that 360, and $360 * 2 = 720$
- Take that 720, and $720 * 1 = 720$

Algorithm

The general form of an accumulator algorithm looks like this:

- *Initialize the accumulator variable*
- *Perform computation*
(e.g. in case of factorial multiply by the next smaller number)
- *Update accumulator variable*
- *Loop until final result is reached*
(e.g. in case of factorial the next smaller number is 1)
- *Output accumulator variable*

Accumulating Results: Factorial

- It looks like we'll need a loop!

```
factorial = 1
for fact in [6, 5, 4, 3, 2, 1]:
    factorial = fact * factorial
```

- Let's trace through it to verify that this works!

Accumulating Results: Factorial

- Why did we need to initialize factorial to 1?
- There are a couple reasons...
 - *Each time through the loop, the previous value of factorial is used to calculate the next value of factorial. By doing the initialization, you know fact will have a value the first time through.*
 - *If you use factorial without assigning it a value, what does Python do?*

Improving Factorial

- What does `range(n)` return?
0, 1, 2, 3, ..., **n-1**
- `range` has another optional parameter:
 - `range(start, n)` returns `start, start + 1, ..., n-1`
 - *E.g. `range(1, 11)` returns: 1,2,3,4,5,6,7,8,9,10*
- But wait! There's more!

`range(start, n, step)`

returns: `start, start+step, ...stopping before n`

- `list(<sequence>)` to make a list

Range()

- Let's try some examples!

```
>>> list(range(10))  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
  
>>> list(range(5,10))  
[5, 6, 7, 8, 9]  
  
>>> list(range(5,10,2))  
[5, 7, 9]
```

range() Forwards or Backwards

- We can do the range for our loop a couple of different ways.
 - *We can count up from 2 to n:*
`range(2, n+1)`
(Why did we have to use n+1?)
 - *We can count down from n to 2:*
`range(n, 1, -1)`

Back at the Factorial Program

Our completed factorial program:

```
#     Program to compute the factorial of a number
#     Illustrates for loop with an accumulator
#     Author: Unit coordinator

def main():
    n = int(input("Please enter an integer: "))
    factorial = 1
    for fact in range(n,1,-1):
        factorial = fact * factorial
    print("The factorial of", n, "is", factorial)
    return

main()
```

Lecture Summary

- We learned how to use the Python math library
- We discussed an example of Quadratic equation
- We discussed an example of Factorial
- We discussed `range()` function



THE UNIVERSITY OF
WESTERN
AUSTRALIA

Lecture 7

Decisions

Objectives of this Lecture

- A little revision
- To understand the conditional (decision) statement *if*
 - *if*
 - *if-else*
 - *Nested if --- elif*
- *Comparison operators*
- *Logical operators*

Revision: Accumulator Algorithm

The general form of an accumulator algorithm looks like:

- *Initialize the accumulator variable*
- *Perform computation*
(e.g. in case of factorial multiply by the next smaller number)
- *Update accumulator variable*
- *Loop until final result is reached*
(e.g. in case of factorial the next smaller number is 1)
- *Output accumulator variable*

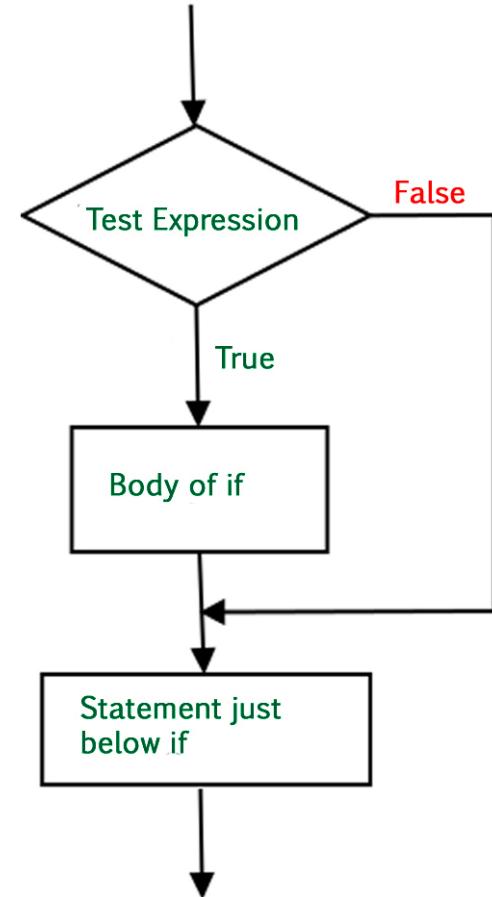
This is called a **Pattern**, or **Software Design Pattern**. That is, a recurring, reusable generalised set of instructions

Decision making



Decision Structures

- So far, we've viewed programs as sequences of instructions that are followed one after the other.
- While this is a fundamental programming concept, it is not sufficient in itself to solve every problem.
- We need to be able to alter the sequential flow of a program to suit a particular situation.

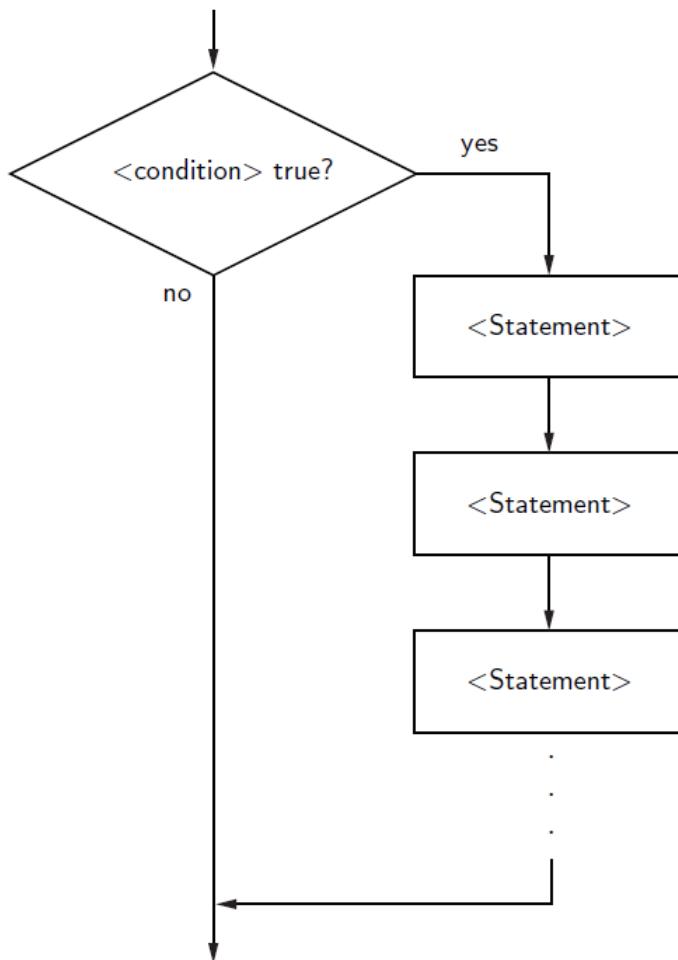


Simple *if* statements

- if <condition> :
<statements to execute if condition is True>
- The condition is a Boolean expression, i.e. evaluates to values True or False
- The condition statement is evaluated;
 - If it evaluates to True, the (indented) statements in the body are executed;
 - Otherwise, execution proceeds to next statement

Don't forget the ":"

Simple *if* statements



Boolean Expressions - Comparisons

- What does a Boolean expression, i.e. condition, look like?
- At this point, let's use simple comparisons.

`<expr> <relop> <expr>`

– *<relop> is short for relational/comparison operator*

Comparison operators

Python	Mathematics	Meaning
<	<	Less than
<=	\leq	Less than or equal to
==	=	Equal to
\geq	\geq	Greater than or equal to
>	>	Greater than
!=	\neq	Not equal to

Note ==

Comparison Operators

```
>>> 8 < 10
```

True

```
>>> 8 > 10
```

False

- Notice the use of == for equality. Since Python uses = to indicate assignment, a different symbol is required for the concept of equality.
- A common mistake is using = in comparisons!

```
>>> 8 == 10
```

False

Forming Comparisons

- When comparing strings, the ordering is *lexicographic*, meaning that the strings are sorted based on the underlying Unicode.
 - *Unicode (and before that, ASCII) is a way of representing characters as integers*
 - *Because of this, all upper-case Latin letters come before lower-case letters. (“Bbbb” comes before “aaaa”)*

```
>>> "Hello" < "hello"
```

True

Logical/Boolean operators

Operation	Meaning
not	Inverse the comparison result e.g. not x return True if x is False or vice versa
and	Returns True only if both inputs are True e.g. x and y return True only when x is True and y is True else it return False
or	Returns False only if both inputs are False e.g. x and y return False only when x is False and y is False else it return True.

Logical operators are used to combine comparisons

Logical operators

- The *and* of two Boolean expressions is True exactly when both of the Boolean expressions are True.
- We can represent this in a *truth table*.
- P and Q represents Boolean expressions.
- Since each Boolean expression has two possible values, there are four possible combinations of values.

P	Q	P and Q
T	T	T
T	F	F
F	T	F
F	F	F

Logical operators

- The `or` of two Boolean expressions is `True` when either Boolean expression is `true`.
- The only time `or` is `false` is when both Boolean expressions are `False`.
- Also, note that `or` is `True` when both Boolean expressions are `True`.

P	Q	$P \text{ or } Q$
T	T	T
T	F	T
F	T	T
F	F	F

Logical operators

- The `not` operator computes the opposite of a Boolean expression.
- `not` is a *unary* operator, meaning it operates on a single Boolean expression.

P	$\text{not } P$
T	F
F	T

- We can put these operators together to make arbitrarily complex Boolean expressions.
- The interpretation of the expressions relies on the precedence rules for the operators.

Example: Temperature Warnings

Design

Input: A value representing a Celsius temperature

Process: (None)

Output:

If temperature greater than 40 print warning

If temperature less than 1 print warning

Simple *if* statements: Example

```
>>> def stayhome():
    temp = float(input("What is the temperature today? "))
    if temp >= 40:
        print("The temperature is too high")
        print("Stay at home")
    if temp <= 0 :
        print("The temperature is too low")
        print("Stay at home")
    print("Have a good day")

>>> stayhome()
```

What is the temperature today? 42

The temperature is too high

Stay at home

Have a good day

What happens for 36?

Simple *if* statements: Example

```
>>> def stayhome():
    temp = float(input("What is the temperature today? "))
    if temp >= 40 or temp <= 0:
        print("The temperature is not appropriate")
        print("Stay at home")
    print("Have a good day")

>>> stayhome()
What is the temperature today? 36
Have a good day
```

Quadratic Equation Example

Consider the quadratic equation program.

```
# quadratic.py
#     A program that computes the real roots of a quadratic equation.
#     Note: This program crashes if the equation has no real roots.

import math # Makes the math library available.

def main():
    print("This program finds the real solutions to a quadratic\n")

    a = float(input("Enter coefficient a: "))
    b = float(input("Enter coefficient b: "))
    c = float(input("Enter coefficient c: "))

    discRoot = math.sqrt(b * b - 4 * a * c)
    root1 = (-b + discRoot) / (2 * a)
    root2 = (-b - discRoot) / (2 * a)

    print("\nThe solutions are:", root1, root2)
```

What does \n do?



Using the Math Library

Running the program

```
>>> main()
This program finds the real solutions to a quadratic
Please enter coefficient a: 3
Please enter coefficient b: 4
Please enter coefficient c: -1

The solutions are: 0.215250437022 -1.54858377035
```

Decisions

Noting the comment, when $b^2 - 4ac < 0$, the program crashes.

```
>>> main()
This program finds the real solutions to a quadratic
Please enter coefficient a: 1
Please enter coefficient a: 2
Please enter coefficient a: 3
```

Traceback (most recent call last):

```
  File "quadratic_roots.py", line 21, in <module>
    main()

  File "quadratic_roots.py", line 15, in main
    discRoot = math.sqrt(b * b - 4 * a * c)

ValueError: math domain error
```

Decisions

We can check for this situation. Here's our first attempt.

```
# quadratic2.py
#     A program that computes the real roots of a quadratic equation.
#     Bad version using a simple if to avoid program crash

import math

def main():
    print("This program finds the real solutions to a quadratic\n")
    a = float(input("Enter coefficient a: "))
    b = float(input("Enter coefficient b: "))
    c = float(input("Enter coefficient c: "))
    discrim = b * b - 4 * a * c
    if discrim >= 0:
        discRoot = math.sqrt(discrim)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2)
```

Two-Way Decisions

- We first calculate the discriminant ($b^2 - 4ac$) and then check to make sure it's non-negative. If it is, the program proceeds and we calculate the roots.
- Look carefully at the program.
 - *What's wrong with it?*
 - *Hint: What happens when there are no real roots?*

Two-Way Decisions

This program finds the real solutions to a quadratic

```
Enter coefficient a: 1
```

```
Enter coefficient b: 1
```

```
Enter coefficient c: 1
```

```
>>>
```

- This is worse than the version that crashes, because we don't know what went wrong!
 - *Don't even know that there is a problem*

Two-Way Decisions

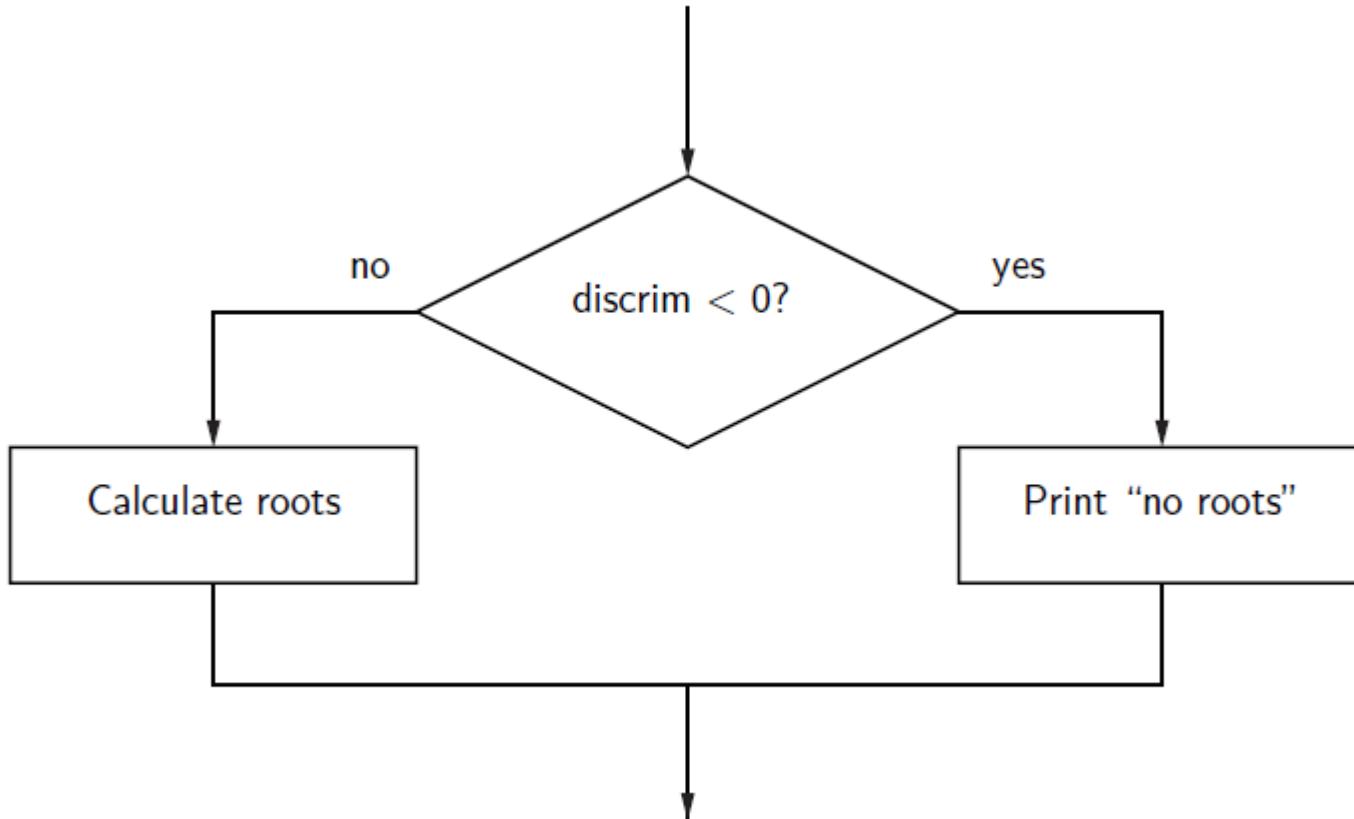
- We could add another if after first if:

```
if discrim < 0:
```

```
    print("The equation has no real roots!")
```

- This works, but feels wrong. We have two decisions, with *mutually exclusive* outcomes
 - *if $discrim \geq 0$ then $discrim < 0$ must be false, and vice versa.*

Two-Way Decisions



Two-Way Decisions

- In Python, a two-way decision can be implemented by attaching an `else` clause onto an `if` clause.
- This is called an `if-else` statement:

```
if <condition>:  
    <statements>  
  
else:  
    <statements>
```

Two-Way Decisions

- When Python encounters **if-else** structure, it first evaluates the condition. If the condition evaluates to True, the statements under the **if** are executed.
- If the condition evaluates to False, the statements under the **else** are executed.
- In either case, the statement following the **if-else** structure is then executed

Two-Way Decisions

```
# quadratic3.py
#     A program that computes the real roots of a quadratic equation.
#     Illustrates use of a two-way decision

import math

def main():
    print "This program finds the real solutions to a quadratic\n"
    a = float(input("Enter coefficient a: "))
    b = float(input("Enter coefficient b: "))
    c = float(input("Enter coefficient c: "))
    discrim = b * b - 4 * a * c
    if discrim < 0:
        print("\nThe equation has no real roots!")
    else:
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print ("\nThe solutions are:", root1, root2 )
```

Two-Way Decisions

This program finds the real solutions to a quadratic

```
Enter coefficient a: 1
```

```
Enter coefficient b: 1
```

```
Enter coefficient c: 2
```

The equation has no real roots!

```
>>>
```

This program finds the real solutions to a quadratic

```
Enter coefficient a: 2
```

```
Enter coefficient b: 5
```

```
Enter coefficient c: 2
```

The solutions are: -0.5 -2.0

Two-Way Decisions

The newest program is great, but it still has some quirks!

This program finds the real solutions to a quadratic

```
Enter coefficient a: 1
```

```
Enter coefficient b: 2
```

```
Enter coefficient c: 1
```

```
The solutions are: -1.0 -1.0
```

Program looks broken, when it isn't

Two-Way Decisions

- While correct, this program output might be confusing for some people.
 - *It looks like it has mistakenly printed the same number twice!*
- A single root occurs when the discriminant is exactly 0, and then the root is $-b/2a$.
- It looks like we need a three-way decision!

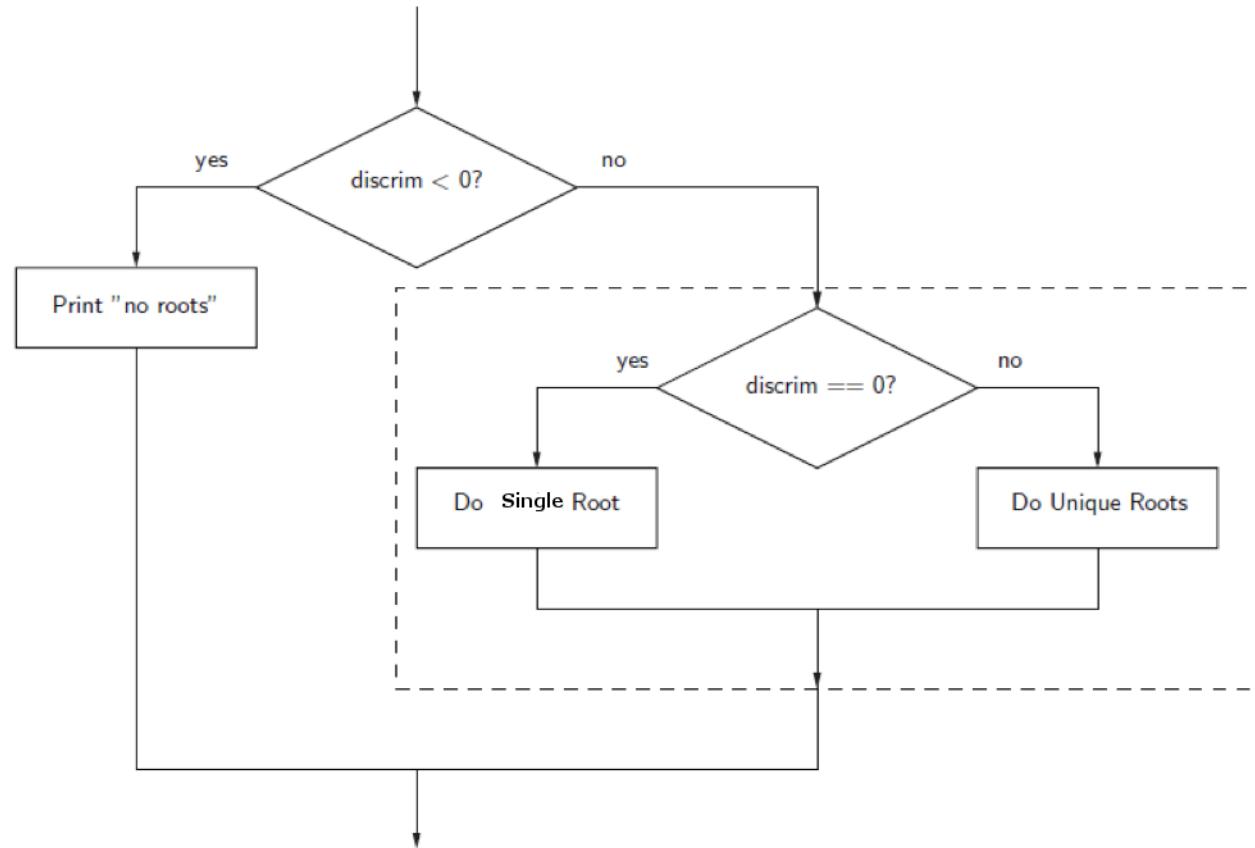
Two-Way Decisions

- Check the value of discrim
 - when < 0 : handle the case of no roots
 - when $= 0$: handle the case of a single root
 - when > 0 : handle the case of two distinct roots
- We can do this with two if-else statements, one inside the other.
- Putting one compound statement inside of another is called *nesting*.

Two-Way Decisions

```
if discrim < 0:  
    print("Equation has no real roots")  
else:  
    if discrim == 0:  
        root = -b / (2 * a)  
        print("There is a single root at", root)  
    else:  
        # Do stuff for two roots
```

Nested Two-Way Decisions



Multi-Way Decisions

- Imagine if we needed to make a five-way decision using nesting. The `if-else` statements would be nested four levels deep!
- There is a construct in Python that achieves this, combining an `else` followed immediately by an `if` into a single `elif`.

Multi-Way Decisions

```
if <condition1>:  
    <statements>  
elif <condition2>:  
    <case2 statements>  
elif <condition3>:  
    <case3 statements>  
...  
else:  
    <default statements>
```

Multi-Way Decisions

- Python evaluates each condition in turn looking for the first one that evaluates to True. If a true condition is found, the statements indented under that condition are executed, and control passes to the next statement after the entire if-elif-else.
- If none are True, the statements under else are performed.
- The final else is optional. If there is no else, it's possible no indented block would be executed.

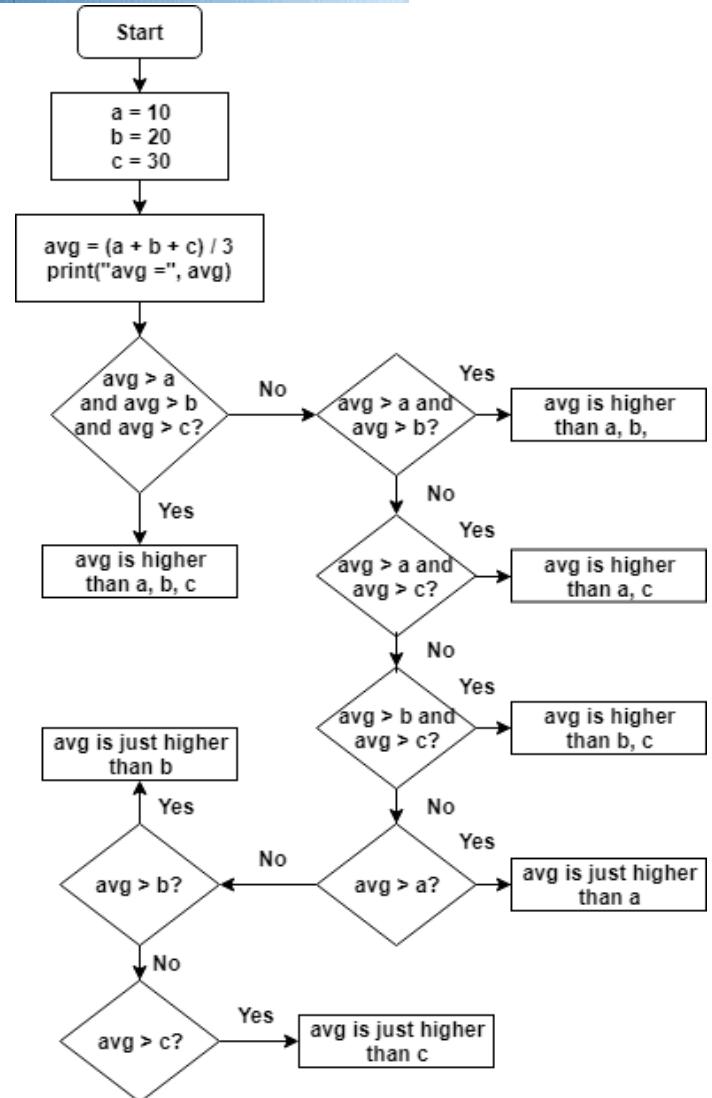
Three-Way Decisions

```
# quadratic4.py
import math
def main():
    print("This program finds the real solutions to a
quadratic\n")

    a = float(input("Enter coefficient a: "))
    b = float(input("Enter coefficient b: "))
    c = float(input("Enter coefficient c: "))
    discrim = b * b - 4 * a * c
    if discrim < 0:
        print("\nThe equation has no real roots!")
    elif discrim == 0:
        root = -b / (2 * a)
        print("\nThere is a single root at", root)
    else:
        discRoot = math.sqrt(discrim)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2 )
```

Nested Two-way Decisions

- What is the purpose of this algorithm?



Anti-bugging

- In the quadratic program we used decision structures to avoid taking the square root of a negative number, thus avoiding a run-time error.
- This is true for many programs: decision structures are used to protect against rare but possible errors.
- Some authors describe this as **anti-bugging**; before processing some data have tests to ensure procedures will be safe.

Lecture Summary

- We learned bout decision making in computer program
- We learned about boolean expressions and their use in **if/if-else/if-elif-else** decision statements.
- We learned about Logical and Comparison operators



THE UNIVERSITY OF
WESTERN
AUSTRALIA

Lecture 7

Functions Part 1

Objectives

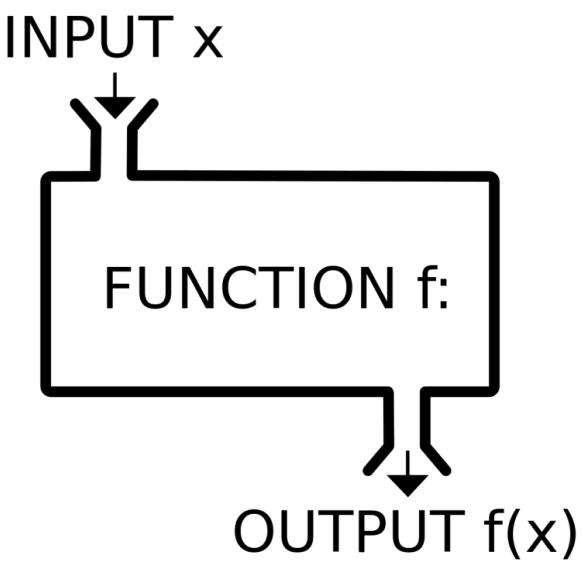
- To understand why programs are divided into sets of cooperating functions.
- To be able to define new functions in Python and write programs that use functions.
- To understand the details of function calls and parameter passing in Python.
- To learn how to pass multiple parameters to functions and get (return) multiple results from functions.
- To understand the relationship between actual and formal parameters.

Why use Functions?

- Having similar code in more than one place has some drawbacks.
 - *Having to type the same code twice or more.*
 - *Unnecessarily complicate the code*
 - *This same code must be maintained in multiple places.
Will differ over time as code maintained*
- Functions are used to:
 - *avoid/reduce code duplication*
 - *make programs easy to understand*
 - *make programs easy to maintain.*

Functions, Informally

- A function is like a **subprogram**, a small program inside a program.
- The basic idea – we write a sequence of statements and then give that sequence a name. We can then 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, Informally

- Happy Birthday lyrics...

```
def main():
    print("Happy birthday to you! ")
    print("Happy birthday to you! ")
    print("Happy birthday, dear Fred... ")
    print("Happy birthday to you!")
```

- Gives us this...

```
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred...
Happy birthday to you!
```

Functions, Informally

- There's some duplicated code in the program!

```
print("Happy birthday to you!")
```

- We can define a function to print out this line:

```
def happy():
    print("Happy birthday to you!")
```

- With this function, we can rewrite our program.

Functions, Informally

- The new program –

```
def singFred():
    happy()
    happy()
    print("Happy birthday, dear Fred...")
    happy()
```

- Gives us this output –

```
>>> singFred()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred...
Happy birthday to you!
```

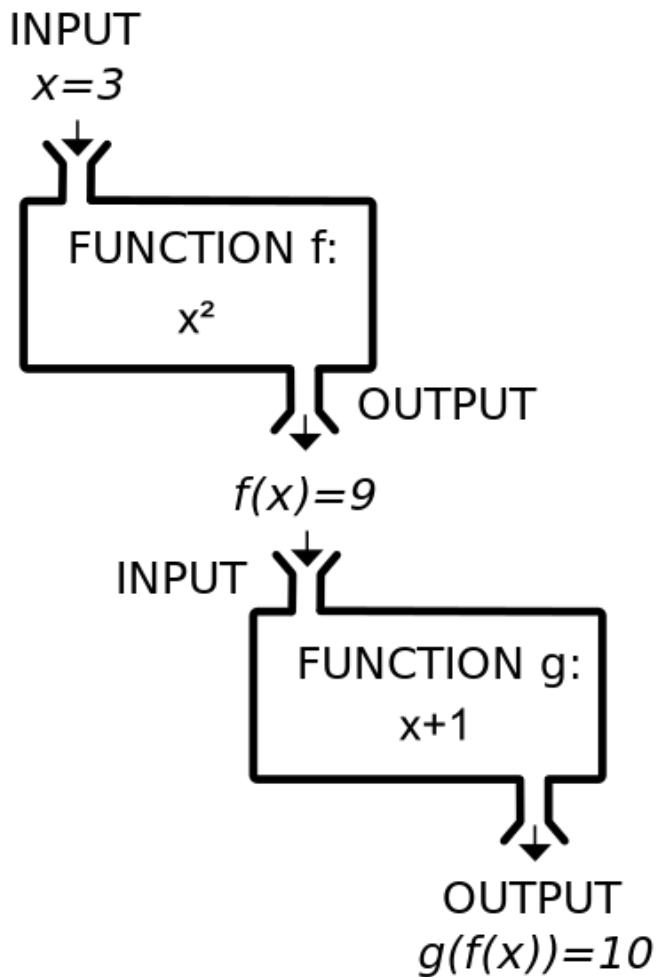
Functions, Informally

- Creating this function saved us a lot of typing!
- What if it's Lucy's birthday? We could write a new singLucy function!

```
def singLucy():  
    happy()  
    happy()  
    print("Happy birthday, dear Lucy...")  
    happy()
```

Functions, Informally

- Multiple functions can be called to do a task
- A problem can be split into multiple parts and each part can be solved by a different team/programmer
- Multiple programmers can work together on bigger projects and share their work in the form of functions



Functions, Informally

- We could write a main program to sing to both Lucy and Fred

```
def main():
    singFred()
    print()
    singLucy()
```

- This gives us this new output

```
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred..
Happy birthday to you!
```

```
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Lucy...
Happy birthday to you!
```

- The only difference is the name in the third print statement.
 - These two routines could be collapsed together by using a **parameter**.
-

Functions, Informally

- The generic function *sing*

```
def sing(person) :  
    happy()  
    happy()  
    print("Happy birthday, dear", person + ".")  
    happy()
```

- This function uses a **parameter** named person.
- A **parameter** is a variable that is initialized when the function is called.
- You have **refactored/generalised** the code

Functions, Informally

- Our new main program:

```
def main():
    sing("Fred")
    print()
    sing("Lucy")
```

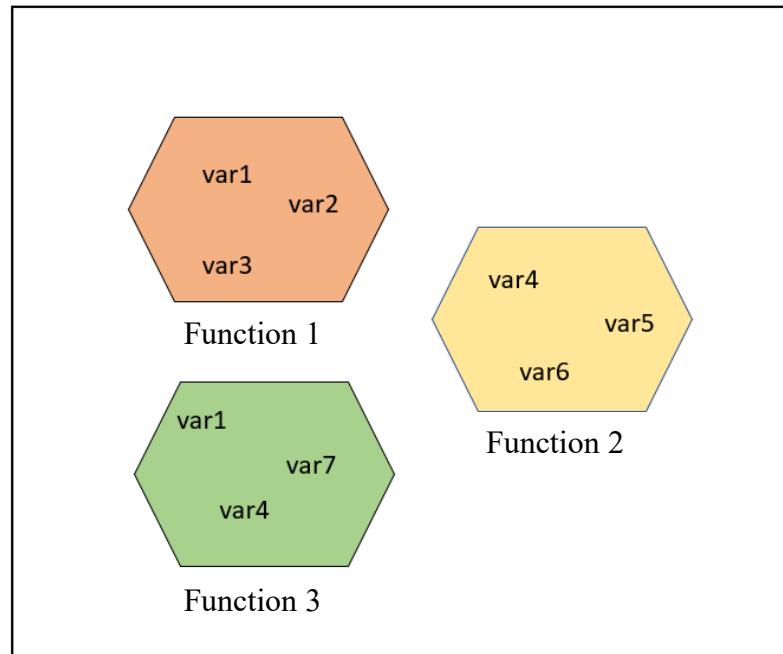
- Gives us this output:

```
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred.
Happy birthday to you!
```

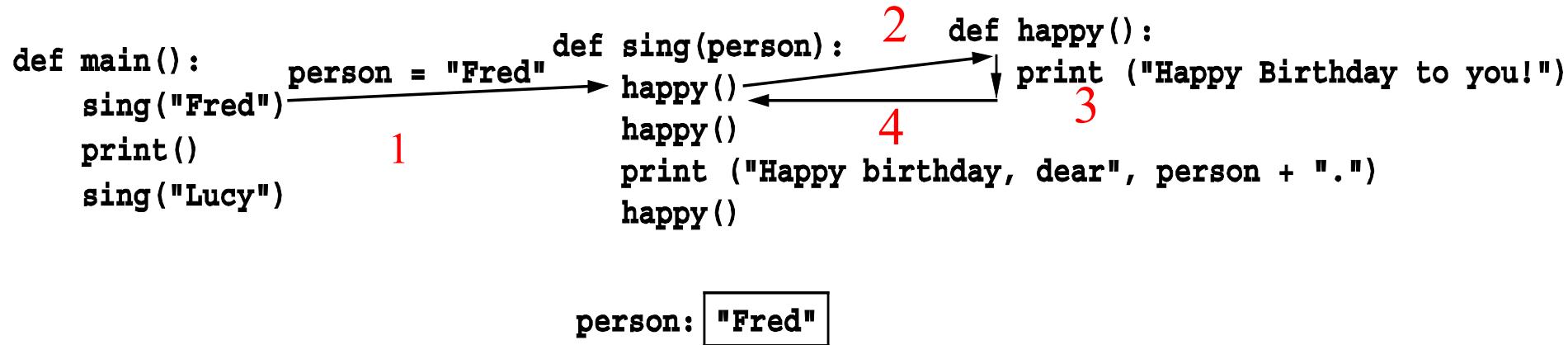
```
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Lucy.
Happy birthday to you!
```

Scope of a Variable

- The *scope* of a variable refers to the places in a program a given variable can be referenced.
- The variables used inside of a function are *local* to that function, even if they happen to have the same name as the variables that appear inside of another function.
- The only way for a function to see a variable from another function is for that variable to be passed as a parameter (and it's the value that is passed!)



Functions and Parameters: The Details



- When Python gets to the end of `sing`, control returns to `main` and continues immediately following the function call.
 - The `person` variable in `sing` disappears when `sing` finishes!
 - Local variables do **not** retain any values from one function execution to the next.
-

Functions and Parameters: The Details

- A function is called by using its name followed by a list of **actual parameters** or **arguments**. <name>(<actual-parameters>)
e.g. `sing("Fred")`
`futureValue(10000, 10, r)`
- When Python comes to a function call, it initiates a four-step process.
 1. *The calling program suspends execution at the point of the call.*
 2. *The formal parameters of the function get assigned the values supplied by the actual parameters in the call.*
 3. *The body of the function is executed.*
 4. *Control returns to the point just after where the function was called.*

Functions and Parameters: The Details

- Let's trace through the following code:

```
sing("Fred")
print()
sing("Lucy")
```

- When Python gets to `sing ("Fred")`, execution of main is temporarily suspended.
- Python looks up the definition of `sing` and sees that it has one formal parameter, `person`.

Passing Multiple Values: Future Value

- To find the future value of an investment, e.g. term deposit, we need three pieces of information.
 - *The principal sum*
 - *The interest rate*
 - *The number of years*
- These three values can be supplied as parameters to the function.

Passing Multiple Values: Future Value

- The resulting function looks like this:

```
def futureValue(p, n, r) :  
    print( p*(1+r)**n)
```

- To use this function, we supply the three values:

```
>>>futureValue(10000, 10, 0.1)  
25937.424601...  
>>>futureValue(10000, 20, 0.15)  
163665.3739294...
```

Functions and Parameters: The Details

- A function definition looks like this:

```
def <name>(<formal-parameters>):  
    <body>
```

- The name of the function must be an identifier
- **Formal-parameters** is a list of variable names. The list can be empty, i.e. just (), if the function does not take any parameters.

Multiple Function Parameters

- Functions can have multiple parameters.
- Formal and actual parameters are matched up based on *position*.
- As an example, consider the call to `futureValue`
- When control is passed to `futureValue`, these parameters are matched up to the formal parameters in the function heading:

```
def futureValue(p, n, r):  
    print( p * (1+r) **n)
```

```
r = 0.1
```

```
>>>futureValue(10000, 10, r)
```

```
25937.424601...
```

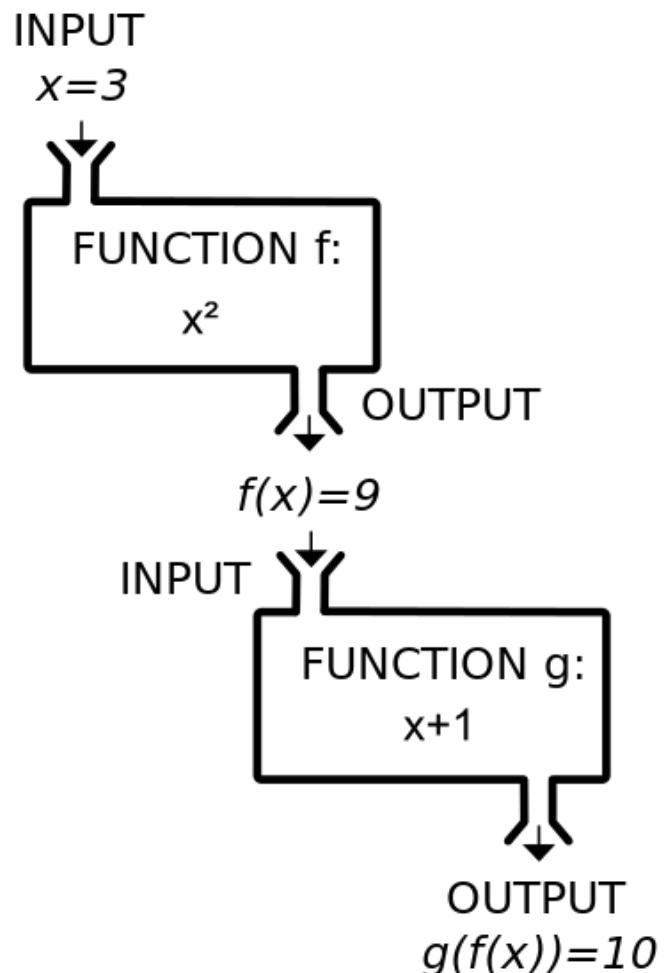
Value of `r` passed to
the parameter `r`
(which could have a
different name!)

Getting Results from a Function

- Passing parameters provides a mechanism for initializing the variables in a function.
- Parameters act as inputs to a function.
- Functions can also return values i.e. functions can have outputs

```
def f(x) :  
    return x*x
```

- 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.



Functions That Return Values

```
def f(x):  
    return x*x  
  
def g(x):  
    return x+1  
  
  
def main():  
    x=3  
    fx=f(x)  
    gx=g(fx)  
    print("Output:", gx)
```

Returning Multiple Values

- Simply list more than one expression in the return statement.

```
def sumDiff(x, y):  
    sum = x + y  
    diff = x - y  
    return sum, diff
```

- When calling this function, use simultaneous assignment.

```
s, d = sumDiff(num1, num2)
```

- The values are assigned based on position, so `s` gets the first value returned (the sum), and `d` gets the second (the difference).

All Functions Return a Value

- 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 mistake is writing a value-returning function but omitting the `return`!
 - *If your value-returning functions produce strange messages, check to make sure you remembered to include the `return`!*

Summary

- We learned how to define new functions and how to call functions in Python.
- We learned how to write programs that use functions to reduce code duplication and increase program modularity.
- We learned how to pass multiple parameters to functions and how to return multiple values from functions.
- We studied the relationship between actual and formal parameters and how functions can change parameters.



THE UNIVERSITY OF
WESTERN
AUSTRALIA

Lecture 8

Strings

Updated COVID-19 pandemic health and safety measures

- Migration to online teaching
 - *Starting Monday 23rd March 2020*
 - *Recorded lectures will be uploaded on LMS*
 - *Consultation hours will remain the same and I will be available via Zoom (details to follow)*
 - *I am happy to add more consultation hours if required*
 - *Labs will be at the same time and lab demonstrators will be available via Zoom (details to follow)*
 - *Physical presence or meeting is not required*
 - *Mid-semester exam may not be held (more details to follow)*
 - *Labs and projects deadlines will remain the same*
- Situation is rapidly changing and University is monitoring it daily. Students will be updated regularly

Adhere to all Federal and State government guidelines about health and safety

Objectives

- To understand the string data type and how strings are represented in a computer.
- To understand the basic idea of sequences and indexing as they apply to Python strings and lists.
- To get familiar with various operations that can be performed on strings through built-in functions and the string library.

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 (').

The String Data Type

```
>>> str1="Hello"  
>>> str2='spam'  
>>> print(str1, str2)  
Hello spam  
>>> type(str1)  
<class 'str'>  
>>> type(str2)  
<class 'str'>
```

The String Data Type

- Getting a string as input

```
>>> firstName = input("Please enter your name: ")  
Please enter your name: John  
>>> print("Hello", firstName)  
Hello John
```

- Notice that the input is not evaluated. We want to store the typed characters, not to evaluate them as a Python expression, e.g. convert to int.

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** (i.e. an integer) determines which character is selected from the string.

The String Data Type

H	e	I	I	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet = "Hello Bob"  
>>> greet[0]  
'H'  
>>> print(greet[0], greet[2], greet[4])  
H l o  
>>> x = 8  
>>> print(greet[x - 2])  
B
```

The String Data Type

H	e	I	I	o		B	o	b
0	1	2	3	4	5	6	7	8

- 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.

```
>>> greet[-1]
```

```
'b'
```

```
>>> greet[-3]
```

```
'B'
```

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 Strings

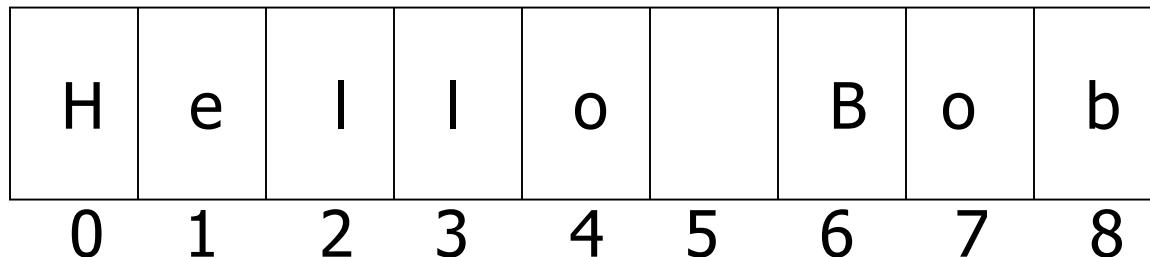


changhsinlee.com

The String Data Type

- Slicing:
 $\langle\text{string}\rangle[\langle\text{start}\rangle:\langle\text{end}\rangle]$
- start and end should both be *ints*
- The slice contains the substring beginning at position start and runs up to **but does NOT include** the position end.

The String Data Type



```
>>> greet[0:3]
```

```
'Hel'
```

```
>>> greet[5:9]
```

```
' Bob'
```

```
>>> greet[:5]
```

```
'Hello'
```

```
>>> greet[5:]
```

```
' Bob'
```

```
>>> greet[:] This is same as greet
```

```
'Hello Bob'
```

The String Data Type

- If either `start` or `end` expression is missing, then the start or the end of the string are used.
- 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 (*)

The String Data Type

Operator	Meaning
+	Concatenation
*	Repetition
<string>[]	Indexing
<string>[:]	Slicing
len(<string>)	Length
for <var> in <string>:	Iteration through characters

The String Data Type

```
>>> "spam" + "eggs"  
'spameggs'  
>>> "Spam" + "And" + "Eggs"  
'SpamAndEggs'  
>>> 3 * "spam"  
'spamspamspam'  
>>> "spam" * 5  
'spamspamspamspamspam'  
>>> (3 * "spam") + ("eggs" * 5)  
'spamspamspameggseggseggseggs'
```

The String Data Type

The function `len()` is used to return the length of string.

```
>>> len("spam")  
4  
>>> for ch in "Spam!":  
    print(ch, end=" ")  
  
S p a m !
```

Note: ' ' printed after each character value

Simple String Processing

- Abbreviations of species names.
 - *In a particular bioinformatics database, names of species are abbreviated to the first 3 letters of the first part (genus) and first 2 letters of the second part (species)*
 - *For example*
 - *Canis familiaris* (dog) ⇒ CANFA
 - *Pan troglodytes* (chimp) ⇒ PANTR

Simple String Processing

```
# get genus and species names
genus = input("Please enter the genus: ")
species = input("Please enter the species: ")

SPECIES_CODE = genus[:3] + species[:2]
# Convert to upper case
SPECIES_CODE = SPECIES_CODE.upper()
```

- We'll be looking at string functions, including `upper()`, later
- This form of functions are called **methods**. Notice variable name followed by dot followed by function call.

String Representation

- 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.
- The mapping of characters to binary codes is arbitrary
 - *as long as everyone uses the **same** mapping.*

String Representation

- 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 characters using 8-bit (1 byte) codes
- Python also supports Unicode which maps 100,000+ characters using variable number of bytes

String Representation

- The `ord` function returns the numeric (ordinal) code of a single character.
- The `chr` function converts a numeric code to the corresponding character.

```
>>> ord("A")
```

```
65
```

```
>>> ord("a")
```

```
97
```

Note that 'A' < 'a'

```
>>> chr(97)
```

```
'a'
```

```
>>> chr(65)
```

```
'A'
```

Programming an Encoder

- Using `ord` and `chr` we can convert a string into and out of numeric form.
- The encoding algorithm is simple:
get the message to encode
for each character in the message:
 print the letter number of the character
- A `for` loop iterates over a sequence of objects, so the `for` loop over a string looks like:
`for <variable> in <string>`

Programming an Encoder

```
# text2numbers.py
#     A program to convert a textual message into a sequence of
#         numbers, utilizing the underlying Unicode encoding.

def main():
    print("This program converts a textual message into a sequence")
    print ("of numbers representing the Unicode encoding of the
message.\n")

    # Get the message to encode
    message = input("Please enter the message to encode: ")

    print("\nHere are the Unicode codes:")
    # Loop through the message and print out the Unicode values
    for ch in message:
        print(ord(ch), end=", ")
    print() # Go to new line at the end of code sequence
```

Replace new line by comma

Programming an Encoder

Shell

```
>>> %Run encoder.py
```

```
This program converts a textual message into a sequence  
of numbers representing the Unicode encoding of the message.
```

```
Please enter the message to encode: fred
```

```
Here are the Unicode codes:  
102,114,101,100,
```

```
>>>
```

Programming an Encoder

```
# A program to convert a textual message into a sequence of
# numbers, utilizing the underlying Unicode encoding.
Improved

def main():
    print("This program converts a textual message into a sequence")
    print("of numbers representing the Unicode encoding of the message.\n")

    message = input("Please enter the message to encode: ")

    print("\nHere are the Unicode codes:")
    for ch in message[:-1]:
        print(ord(ch), end=", ")
    print(ord(message[-1]))
```

Programming a Decoder

- We now have a program to convert messages into a type of “code”, but it would be nice to have a program that could decode the message!
- The outline for a decoder:

get the sequence of numbers to decode

message = ""

for each number in the input:

 convert the number to the appropriate character

 add the character to the end of the message

print the message

Programming a Decoder

- The variable message is an accumulator variable, initially set to the **empty string**, the string with no characters ("" or '').
- Each time through the loop, a number from the input is converted to the appropriate character and appended to the end of the accumulator.
- How do we get the sequence of numbers to decode?
- Read the input as a single string, then split it apart into substrings, each of which represents one number.

Programming a Decoder

- The new algorithm:

*get the sequence of numbers as a string, inString
split inString into a sequence of small strings (of digits)
message = "" # Empty string
for each of the smaller strings:
 change the digits into the number they represent
 append the ASCII character for that number to message
print message*

string → list of digit strings → list of numbers → string

“102,114” → [“102”, “114”] → [102, 114] → “fr”

Programming a Decoder

- Strings have useful methods associated with them
- One of these methods is split. This will split a string into substrings based on a separator, e.g space.

```
>>> "Hello string methods!".split()  
['Hello', 'string', 'methods!']
```

This is a list.
More about
them in the
next lecture

- This is the same as:

```
>>> a = "Hello string methods!"  
>>> a.split()
```

Programming a Decoder

- Split can use other separator characters other than space, by supplying the character as a parameter.

```
>>> "32,24,25,57".split(",")
['32', '24', '25', '57']
```

Programming a Decoder

```
# numbers2text.py
#     A program to convert a sequence of Unicode numbers into
#             a string of text.

def main():
    print ("This program converts a sequence of Unicode numbers into")
    print ("the string of text that it represents.\n")

    # Get the message to encode
    inString = input("Please enter the Unicode-encoded message: ")

    # Loop through each substring and build Unicode message
    message = ""
    for numStr in inString.split(','):
        codeNum = int(numStr)      # convert the (sub)string to a number
        # append character to message
        message = message + chr(codeNum)

    print("\nThe decoded message is:", message)
```

Programming a Decoder

- The `split` function produces a sequence of strings.
- Each time through the loop, the next substring is:
 - *Assigned to the variable* `numStr`
 - *Converted to the appropriate Unicode character*
 - *Appended to the end of message.*

Programming a Decoder

Shell

```
>>> %Run encoder.py
```

This program converts a textual message into a sequence of numbers representing the Unicode encoding of the message.

Please enter the message to encode: fred

Here are the Unicode codes:
102,114,101,100,

```
>>> %Run decoder.py
```

This program converts a sequence of Unicode numbers into the string of text that it represents.

Please enter the Unicode-encoded message: 102,114,101,100

The decoded message is: fred

```
>>>
```

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 through an insecure medium e.g. Internet.
- The Unicode mapping between character and number is an industry standard, so it's not “secret”.

Encryption : Substitution Cipher

- In a simplistic way, if we replace the Unicode with some other code (that is only known to the sender and receiver) we will achieve encryption.
- This is called *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.

More String Methods

- There are a number of other string methods. Try them all!
 - `s.tr()` – *Return a string representation*
 - `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*

More String Methods

- `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*
- `s.rjust(width)` – *Like ljust, but s is right-justified*

More String Methods

- `s.lower()` – *Copy of s in all lowercase letters*
- `s.lstrip()` – *Copy of s with leading whitespace removed*
- `s.replace(olgsub, newsub)` – *Replace occurrences of oldsub in s with newsub*
- `s.rfind(sub)` – *Like find, but returns the right-most position*

More String Methods

- `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*

Summary

- We learned how strings are represented in a computer.
- We learned how various operations can be performed on strings.



Lecture 9

Strings and Lists are Sequences

Objectives

- To understand the list data type and how strings and lists are subclasses of sequences
- To understand the differences between mutable and immutable sequences.
- To get familiar with various operations that can be performed on lists through built-in functions.

Revision: 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 (').

Revision: Indexing

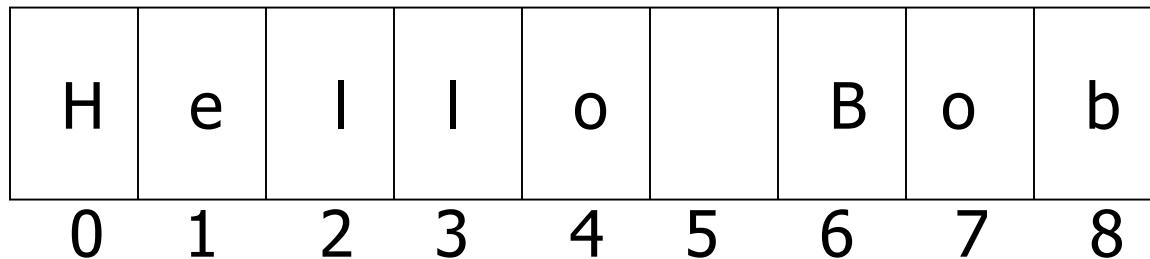
- We can access the individual characters in a string through **indexing**.
- The positions in a string are numbered from the left, starting with 0.
- The general form is `<string> [<expr>]` where the value of `expr` (i.e. an integer) determines which character is selected from the string.

Revision: Indexing

H	e	I	I	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet = "Hello Bob"  
>>> greet[0]  
'H'  
>>> print(greet[0], greet[2], greet[4])  
H l o  
>>> x = 8  
>>> print(greet[x - 2])  
B
```

Revision: Slicing



```
>>> greet[0:3]
```

'Hel'

```
>>> greet[5:9]
```

' Bob'

```
>>> greet[:5]
```

'Hello'

```
>>> greet[5:]
```

' Bob'

```
>>> greet[:] This is same as greet
```

'Hello Bob'

Int to Month

- Converting an int that stands for a month into the three letter abbreviation for that month.
- Store all the names in one big string:

months= "JanFebMarAprMayJunJulAugSepOctNovDec"

- Use the month number as an index for slicing this string:

```
pos *= 3  
monthAbbrev = months [pos:pos+3]      Still not right
```

Int to Month

```
# month.py
# A program to print the abbreviation of a month, given its number

def main():
    # months is used as a lookup table
    months = "JanFebMarAprMayJunJulAugSepOctNovDec"
    n = int(input("Enter a month number (1-12): "))

    # compute starting position of month n in months
    pos = (n-1) * 3

    # Grab the appropriate slice from months
    monthAbbrev = months[pos:pos+3]

    # print the result
    print ("The month abbreviation is", monthAbbrev + ".")
```

Int to Month

```
>>> main()  
Enter a month number (1-12): 1  
The month abbreviation is Jan.  
>>> main()  
Enter a month number (1-12): 12  
The month abbreviation is Dec.
```

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

Lists as Sequences

- Strings are always sequences of characters, but **lists** can be sequences of arbitrary values.
- Lists can have numbers, strings, or both!

```
myList = [1, "Spam ", 3.142, True]
```

Lists as Sequences

- 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:

```
months = ["Jan", "Feb", "Mar", "Apr",
          "May", "Jun", "Jul", "Aug",
          "Sep", "Oct", "Nov", "Dec"]
```

- Note that the months line overlaps a line. Python knows that the expression isn't complete until the closing] is encountered.

Lists as Sequences

- To get the months out of the sequence, do this:

```
monthAbbrev = months[n-1]
```

Rather than this:

```
monthAbbrev = months[pos:pos+3]
```

Lists as Sequences

```
# month2.py
# A program to print the month name, given it's number.
# This version uses a list as a lookup table.

def main():

    # months is a list used as a lookup table
    months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
              "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]
    n = int(input("Enter a month number (1-12) : "))
    print ("The month abbreviation is", months[n-1] + ".") .
```



Note: Since the list is indexed starting from 0, the $n-1$ calculation is straight-forward enough to put in the print statement without needing a separate step.

Lists as Sequences

- This version of the program is easy to extend to print out the whole month name rather than an abbreviation

```
months = ["January", "February", "March",
          "April", "May", "June", "July",
          "August", "September", "October",
          "November", "December"]
```

Lists as Sequences

- It turns out that strings are really a special kind of **sequence**, so these operations also apply to other sequences, particularly **lists**.

```
>>> [1,2] + [3,4] # + is concatenate, or vector addition!
[1, 2, 3, 4]
>>> [1,2]*3      # This is NOT scalar multiplication!
[1, 2, 1, 2, 1, 2]
>>> grades = ['HD', 'D', 'Cr', 'P', 'N']
>>> grades[0]
'HD'
>>> grades[2:4]
['Cr', 'P']
>>> len(grades)
```

Lists as Sequences

- Lists are **mutable**, meaning they can be changed.
Strings can **not** be changed.

```
>>> myList = [34, 26, 15, 10]
>>> myList[2]
15
>>> myList[2] = 0
>>> myList
[34, 26, 0, 10]
>>> myString = "Hello World"
>>> myString[2]
'1'
>>> myString[2] = "p"
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    myString[2] = "p"
TypeError: object doesn't support item assignment
```

Lists Have Methods, Too

- Back at the decoder program

```
# Loop through each substring and build Unicode message
message = ""
for numStr in inString.split(','):
    codeNum = int(numStr)      # convert the (sub)string to a number
    # append character to message
    message = message + chr(codeNum)
```

- Each iteration a copy of the message so far is created and another character tacked onto the end. *New string*
- As we build up the message, we keep recopying a longer and longer string just to add a single character at the end!

Lists Have Methods, Too

- We can avoid this recopying by creating a list of characters and then using `append()`
 - *each new character is added to the end of the existing list.*
- Since lists are mutable, the list is changed “in place” without having to copy the content over to a new object.
- When done, we can use `join()` to concatenate the characters into a string.

Lists Have Methods, Too

```
# numbers2text2.py
#
#     A program to convert a sequence of Unicode numbers into
#             a string of text. Efficient version using a list accumulator.

def main():
    print("This program converts a sequence of Unicode numbers into")
    print("the string of text that it represents.\n")
    # Get the message to encode
    inString = input("Please enter the Unicode-encoded message: ")
    # Loop through each substring and build Unicode message
    chars = []
    for numStr in inString.split():
        codeNum = int(numStr)                      # convert digits to a number
        chars.append(chr(codeNum))                  # accumulate new character

    message = "".join(chars)          # join with empty string separator
    print("\nThe decoded message is:", message)
```

Lists Have Methods, Too

- List specific functions
 - `min()` # Watch out for mixed types
 - `max()`
 - `list()` – convert sequence into a list
 - `append()` – Add in place to list
 - `reverse()` – Reverse in place a list
 - Strings and lists share sequence functions
 - `len()`
 - `+` (concatenation)
 - Slicing using `[:]`
 - `in` (but just letters in strings)
-

Summary

- We have learned that strings and lists are just different sorts of sequences
- Lists have fewer restrictions than strings
- Many (not all) of the operations that work on strings also work on lists. There are also list specific functions.



Lecture 10

File Processing

Objectives

- To understand how to format strings
- To understand the basic text file processing concepts in Python.
- To learn how to read and write text files in Python and string formatting

Revision

```
intlist = []
for i in range(6) :
    if i % 2 == 0 :
        intlist.append(i)
    else:
        intlist[-1] += 1
print(intlist)
```

- What is printed when the code is executed

String Formatting

```
>>> amount = 1.50  
>>> print(amount)  
1.5
```

- If the code above is meant to represent an amount in dollars and cents, we conventionally do not use fractional dollars but rather dollars with 2 digits for cents
- Use the format method

```
>>> print("${0:0.2f} change".format(amount))  
$1.50 change
```

- The first part is the string to be printed, called the **template string**. The part between {} is the **format specifier** (where the value is to be inserted and how it should look).

String Formatting

```
"${0:0.2f} change".format(amount)
```

- The template contains a single specifier slot with the description: `0:0.2f`
- Form of description:
`<index>:<format-specifier>`
- **Index** tells which parameter to insert into the slot (there can be more than one). In this case, `amount` (numbered from 0!)

String Formatting

Looking at 0.2f

- The formatting specifier has the form:
`<width>.<precision><type>`
- `f` means "fixed point" number
- `<width>` tells us how many spaces to use to display the value. 0 means to use as much space as necessary.
- `<precision>` is the number of decimal places.

```
>>> "Compare {0} and {0:0.20f}".format(3.14)  
'Compare 3.14 and 3.140000000000001243'
```

String Formatting Example

```
# Print out a child's multiplication table 0..10
def multiplication_table() :
    for i in range(11) :
        for j in range(11) :
            print("{0:0d} x {1:0d} = {2:0d}".format(i, j, i*j))
    print()

>>> %Run multiplication_table.py
0 x 0 = 0
0 x 1 = 0
0 x 2 = 0
0 x 3 = 0
0 x 4 = 0
0 x 5 = 0
0 x 6 = 0
0 x 7 = 0
0 x 8 = 0
0 x 9 = 0
0 x 10 = 0

1 x 0 = 0
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
```

A basic child's multiplication table to illustrate string formatting.

Needs modification to properly resemble table, e.g. numbers along top and down left hand side

Multiline Strings

- You sometimes need strings that span more than one line. Two ways to do this:
- Embedded '\n' in single string
 - "Twas brillig, and the slithy toves \n Did gyre
and gimble in the wabe \n All mimsy were the
borogoves \n And the mome raths outgrabe."

- Multiline string:

```
"""Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
The frumious Bandersnatch!"""
```

- """ also available
 - "hello\tworld"
 - '\t' is the tab character
-

Files: Multi-line Strings

- A **file** is a sequence of data that is stored in secondary memory (disk drive).
 - *Files don't disappear when program ends*
- 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.

File Processing

- The process of *opening* a file involves associating a file on disk with variable in memory.
- We can manipulate the file by manipulating this variable.
 - *Read from the file*
 - *Write to the file*

File Processing

- When you've finished working 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.
 - *Think of safely ejecting your memory stick*

File Processing - Reading

- Reading a file into a program, e.g. word processor
 - *File opened*
 - *Contents read into RAM*
 - *File closed*
 - *Changes to the file are made to the copy stored in memory, not on the disk.*

File Processing - Writing

- Saving a file, i.e. data in RAM onto file
 - *The original file on the disk is reopened in a mode that will allow writing (this actually erases the old contents unless specifically appending)*
 - *File writing operations copy the version of the document in memory to the disk*
 - *The file is closed*

File Processing in Python

- 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")`

File Methods

- `<file>.read()` – returns the entire remaining contents of the file as a single (possibly large, multi-line) string. Watch out for final `\n`
- `<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.

File Processing

```
# printfile.py
#           Prints a file to the screen.

def main():
    fname = input("Enter filename: ")
    infile = open(fname, 'r')
    data = infile.read()
    infile.close()
    print(data)
```

- First, prompt the user for a file name
 - Open the file for reading
 - The file is read as one string and stored in the variable data
-

File Processing

- `readline` can be used to read the next line from a file, including the trailing newline character

```
infile = open(someFile, "r")
for i in range(5):
    line = infile.readline()
    print(line[:-1])
```

- This reads the first 5 lines of a file
- Slicing is used to strip out the newline characters at the ends of the lines

File Processing Loop

- Python treats the file itself as a sequence of lines!

```
infile = open(someFile, "r")
for line in infile:
    # process the line here
infile.close()
```

- Most efficient way to read through (and process) file
 - *Multiple calls to readline() is inefficient*

File Processing

- Opening a file for writing prepares the file to receive data
- If you open an existing file for writing, you wipe out the file's contents. If the named file does not exist, a new one is created.

```
outfile = open("mydata.out", "w")  
outfile.write(<string>)
```

May use `writelines()` for writing sequence (list) of strings

Example Program: Batch Usernames

- **Batch mode processing** is where program input and output are done through files (the program is not designed to be interactive)
 - *Real strength of Python of many applications.
GUI is fine for small number of cases, but need automation for larger number.*
- Let's create usernames for a computer system where the first and last names come from an input file.

Example Program: Batch Usernames

```
# userfile.py
#     Program to create a file of usernames in batch mode.

def main():
    print ("This program creates a file of usernames from")
    print ("a file of names.")

    # get the file names
    infilename = input("Which file are the names in? ")
    outfileName = input("Where should the usernames go? ")

    # open the files
    infile = open(infilename, 'r')
    outfile = open(outfile, 'w')
```

Example Program: Batch Usernames

```
# process each line of the input file
for line in infile:
    # get the first and last names from line
    first, last = line.split()
    # create a username
    uname = (first[0]+last[:7]).lower()
    # write it to the output file
    outfile.write(uname)

# close both files
infile.close()
outfile.close()

print("Usernames written to", fileName)
```

Example Program: Batch Usernames

- Things to note:
 - *It's not unusual for programs to have multiple files open for reading and writing at the same time. However, if a file is no longer needed, close it as there is a limit to number of open files.*
 - *The lower method is used to convert the names into all lower case, in the event the names are mixed upper and lower case, e.g de Witt.*

Summary

- We learned how to format a string for output that is more readable and looks nice
- We learned how to read files
 - *All at once*
 - *Single lines*
 - *All the lines, line-by-line*
- We learned how to write into files



THE UNIVERSITY OF
WESTERN
AUSTRALIA

Lecture 11

Functions Part 2

Objectives

- To round out the discussion of functions.
- To learn how functions can change parameters.

Revision on Functions

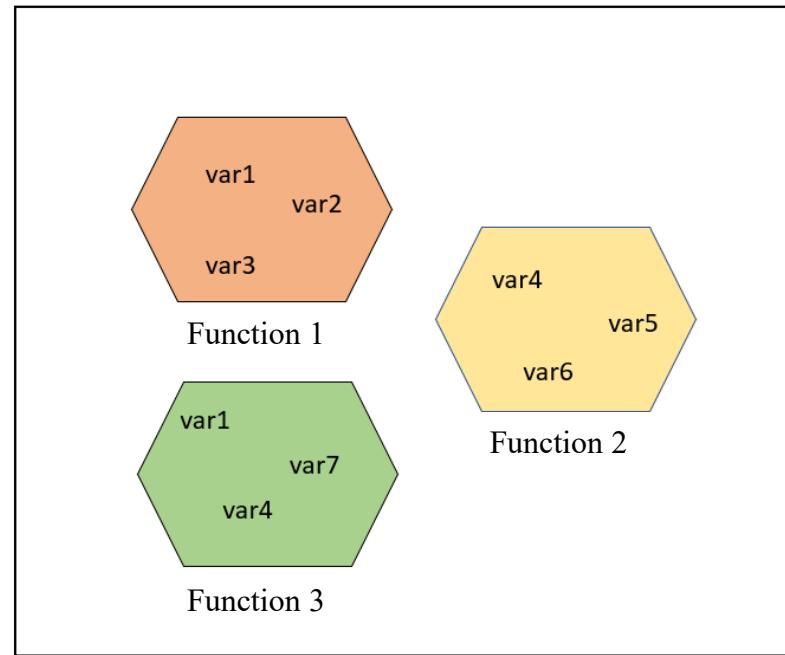
- Some of our programs comprised a single function called `main()`.
- We have already used built-in Python functions e.g. `abs()`, `int()`, `input()`, `print()` etc.
- We have used functions from the standard libraries e.g. `math.sqrt()`
- You may have defined your own functions during the labs, and Project, for answering different questions

Revision: Why use Functions?

- Having similar code in more than one place has some drawbacks.
 - *Having to type the same code twice or more.*
 - *Unnecessarily complicate the code*
 - *This same code must be maintained in multiple places. Will differ over time as code maintained*
- Functions are used to:
 - *avoid/reduce code duplication*
 - *make programs easy to understand*
 - *make programs easy to maintain.*

Revision: Scope of a Variable

- The *scope* of a variable refers to the places in a program a given variable can be referenced.
- The variables used inside of a function are *local* to that function, even if they happen to have the same name as the variables that appear inside of another function.
- The only way for a function to see a variable from another function is for that variable to be passed as a parameter.



Functions that Modify Parameters

- Return values are the main way to send information from a function back to the caller.
- However, in certain circumstances we can communicate back to the caller **by making changes to the function parameters.**
- Understanding when and how this is possible requires the mastery of some subtle details about how assignment works and the relationship between actual and formal parameters.

Functions that Modify Parameters

- Suppose you are writing a program that manages bank accounts and one of the functions accumulates interest on the account.

```
def addInterest(balance, rate):  
    newBalance = balance * (1 + rate)  
    balance = newBalance
```

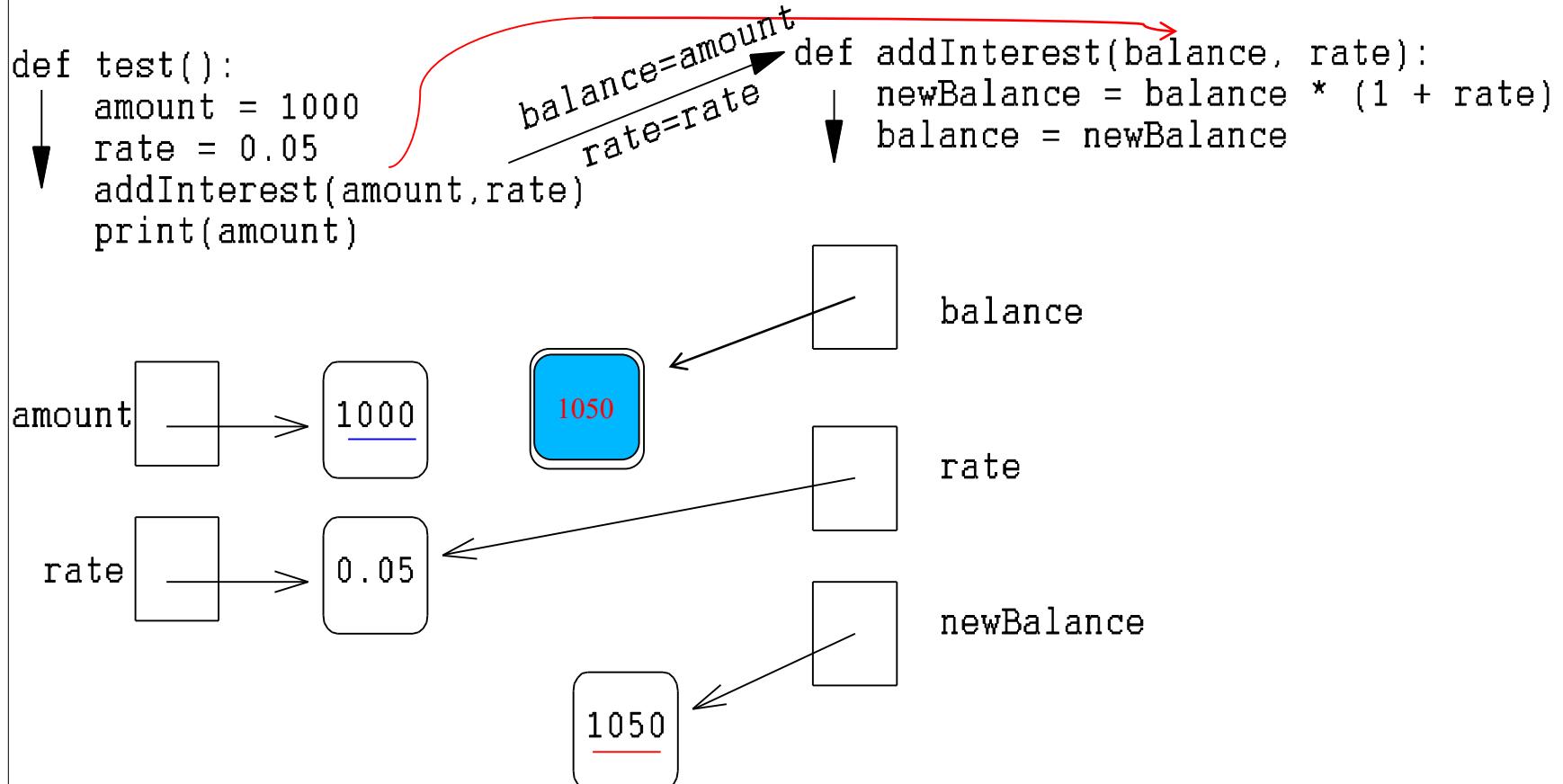
- Let's write a main program to test this:

```
def test():  
    amount = 1000  
    rate = 0.05  
    addInterest(amount, rate)  
    print(amount)  
  
>>> test()  
1000
```

Is this a mistake? **NO**

Functions that Modify Parameters

```
def test():
    amount = 1000
    rate = 0.05
    addInterest(amount,rate)
    print(amount)
```



The value of `amount` passed to `balance`. New value of `balance` computed, but not reflected back

Functions that Modify Parameters

- To summarize: the formal parameters of a function only receive the *values* of the actual (calling) parameters. The function does not have access to the calling variable.
- 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**.

Functions that Modify Parameters

Since Python doesn't pass arguments by-reference, one alternative is to change the addInterest function so that it returns newBalance.

```
def addInterest(balance, rate):
    newBalance = balance * (1 + rate)
    return newBalance

def test():
    amount = 1000
    rate = 0.05
    amount = addInterest(amount, rate)
    print(amount)

>>> test()
1050
```

Functions that Modify Parameters

- Suppose we are writing a program that deals with many accounts.
 - *We could store the account balances in a list, then add the accrued interest to each of the balances in the list.*
- We could update the first balance in the list with code like:

```
balances[0] = balances[0] * (1 + rate)
```
- This code says, “multiply the value in the 0th position of the list by (1 + rate) and store the result back into the 0th position of the list.”
- A more general way to do this would be with a loop that goes through positions 0, 1, ..., length – 1.

Functions that Modify Parameters

```
# addinterest3.py
#     Illustrates modification of a mutable parameter (a list).

def addInterest(balances, rate):
    for i in range(len(balances)):
        balances[i] = balances[i] * (1+rate)

def test():
    amounts = [1000, 2200, 800, 360]
    rate = 0.05
    addInterest(amounts, 0.05)
    print(amounts)

test()
[1050.0, 2310.0, 840.0, 378.0]
```

But Python is Pass
by Value!
What is happening?!

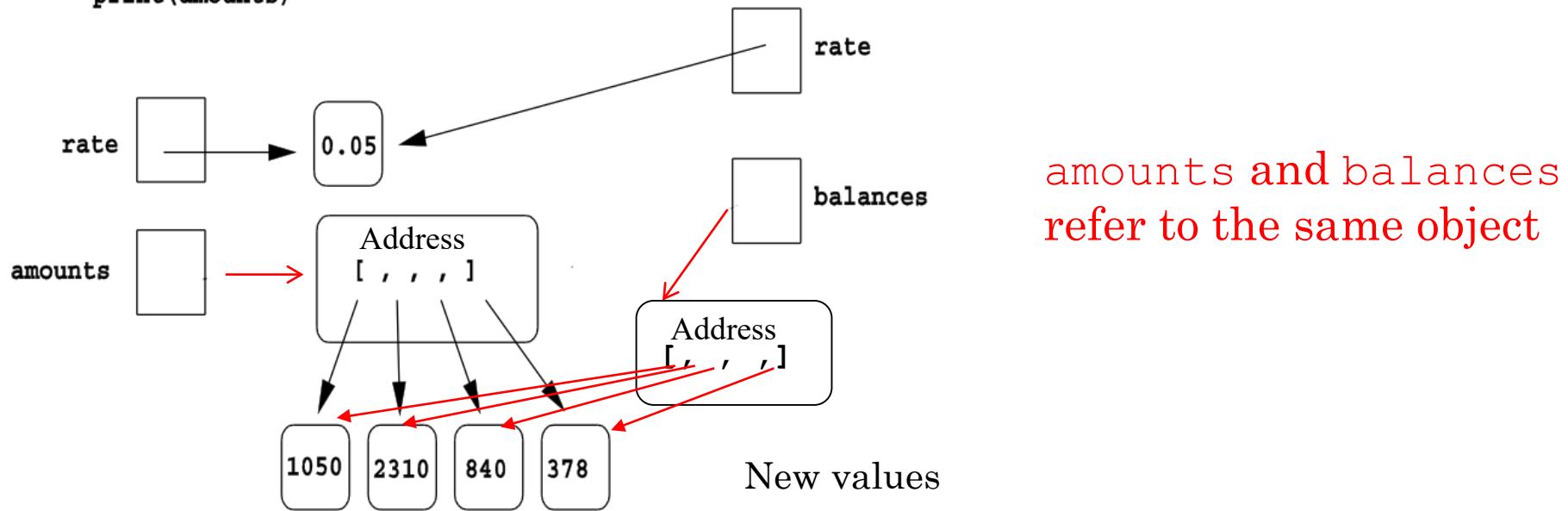
Functions that Modify Parameters

- When `addInterest` terminates, the list stored in `amounts` contains the new values.
- The variable `amounts` wasn't changed (it's the same list), but the contents of that list has changed, and this change is visible to the calling program.
- Parameters are always passed by value. However, if the value of the variable is a mutable object (like a list or other objects), then changes to the internal state of the object ***will*** be visible to the calling program.

Functions that Modify Parameters

```
def test():
    amounts = [1000,2150,800,3275]
    rate = 0.05
    addInterest(amounts,rate)
    print(amounts)
```

```
def addInterest(balances, rate):
    for i in range(len(balances)):
        balances[i] = balances[i] * (1+rate)
```



Functions that Modify Parameters



Two names: Nanga Parbat, Killer Mountain (Himalayas, Pakistan)

https://en.wikipedia.org/wiki/Nanga_Parbat (9th highest mountain in world)

Two identifies for the same object – just the contents have been changed

Default Values for Parameters

- The most common way to call functions is to provide N values for N parameters.
- However, sometimes it's handy to be able to ignore less important parameters and just have default values.
- For example, you wish to define the function `mean()`, but offer a range of different interpretation of mean, e.g. arithmetic (i.e. standard), geometric mean and harmonic mean. The function definition could begin:

```
def mean(values, type="arithmetic") :  
    if type == "arithmetic" :  
        .....  
    elif type == "geometric" :  
        .....
```

Default Values for Parameters

- `mean([1, 2, 3, 4, 5])` is the same as calling `mean([1, 2, 3, 4, 5], "arithmetic")`, but if you want the geometric mean, that has to be called explicitly, `mean([1, 2, 3, 4, 5], "geometric")`
- One Gotcha. The parameters with default values have to come **after** the positional parameters.
- Upside: Only the important parameters (which come first) need be specified

Functions and Program Structure

- So far, functions have been used as a mechanism for reducing code duplication.
- 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 is to make your programs more **modular**.
 - *Recall problem decomposition*
 - *One way to deal with this complexity is to break an algorithm down into smaller subprograms, each of which makes sense on its own.*
 - *Separation of concerns*

Functions and Program Structure

- For example, a function at the start can deal with user input data
 - *Check that inputs are of the expected type and range.*
 - *Once the input data is checked and known to be sound, another function (set of functions) can process the data*

Summary

- We learned the advantages of dividing a program into multiple cooperating functions.
- We studied how functions can change parameters.



THE UNIVERSITY OF
WESTERN
AUSTRALIA

Lecture 12

Loops

Objectives

- To understand the concepts of definite (`for`) and indefinite (`while`) loops.
- To understand interactive loop and sentinel loop and their implementations using a `while` statement.
- To be able to design and implement solutions to problems involving loop patterns including nested loop structures.

for Loop: Revision

```
for i in range(10):  
    # do something  
#-----  
myList = [2,3,4,9,10]  
for x in myList:  
    # iterates through the list elements  
    # do something that involves the list elements  
#-----  
myString = "hello there, hello world!"  
for ch in myString:  
    # iterates through the string characters  
#-----  
infile = open(someFile, "r")  
for line in infile:  
    # iterate through the lines of the file  
infile.close()
```

Indefinite Loops

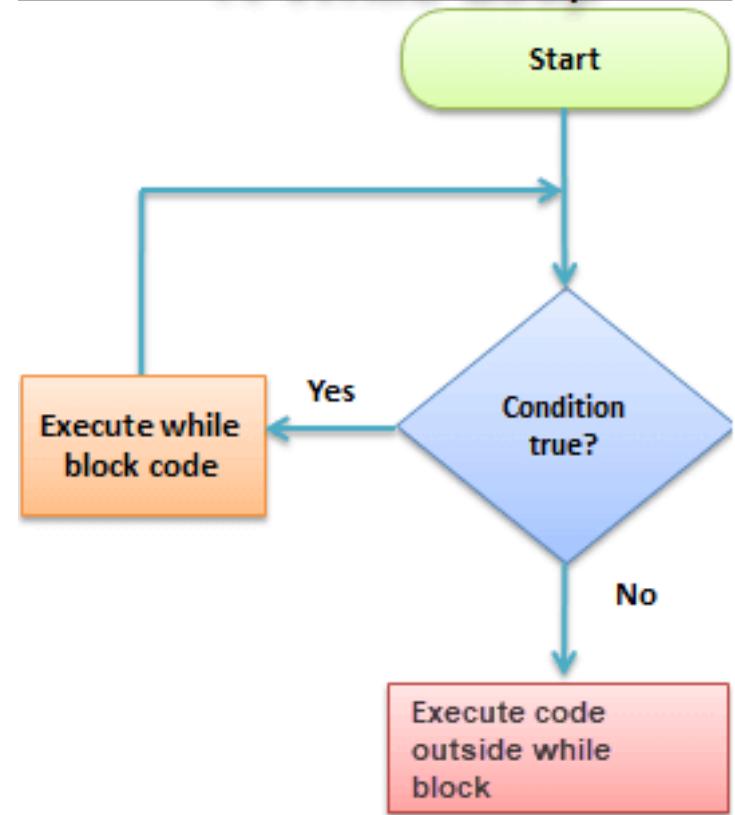
- Definite loops can be used only if we know the number of iterations ahead of time, i.e. before the loop starts.
- Sometimes, we don't know how many iterations we need until all the data has been entered.
- The **indefinite or conditional loop** keeps iterating until certain conditions are met.

Indefinite Loops

- `while <condition>:
 <body>`
- `<condition>` is a Boolean expression, just like in `if` statements. `<body>` is a sequence of one or more statements.
- Semantically, the body of the loop executes repeatedly as long as the condition remains true.
- When the condition is false, the loop terminates.

Indefinite Loops

- The condition is tested at the top of the loop.
- This is known as a **pre-test** loop.
- If the condition is initially false, the loop body will not execute at all.



Indefinite Loops

- Example of a `while` loop that counts from 0 to 9:

```
i = 0
while i < 10: # valid but poor use of while
    print(i)
    i += 1
```

- The code has the same output as this `for` loop:

```
for i in range(10) :# this is the right way
    print(i)
```

- The `while` loop requires us to manage the loop variable `i` by initializing it to 0 before the loop and incrementing it at the bottom of the body.
- In the `for` loop this is handled automatically.

Indefinite Loops

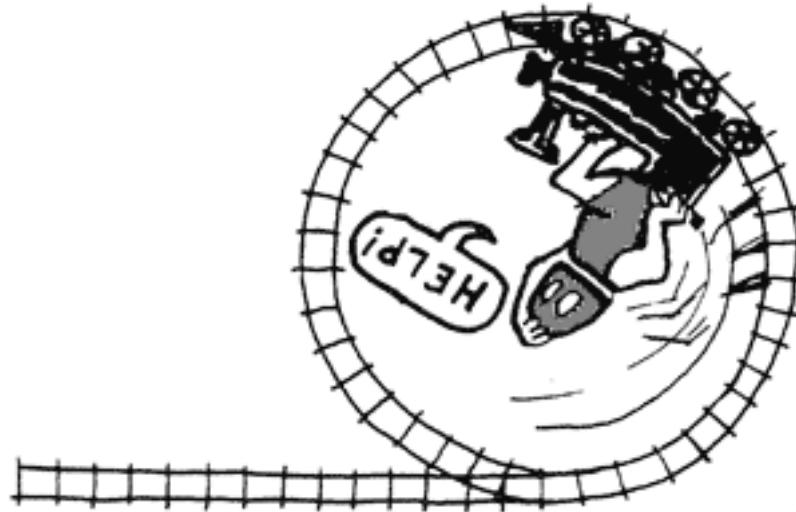
- The `while` statement is simple, but yet powerful and dangerous – they are a common source of program errors.

```
i = 0
while i < 10:
    print(i)
```

- What happens with this code?
- The value of `i` never changes inside the loop body.
- This is an example of an **infinite loop**.

Getting out of an Infinite Loop

- What should you do if you're caught in an infinite loop?
 - *First, try pressing control-c (or STOP on Thonny)*
 - *If that doesn't work, try control-alt-delete*
 - *If that doesn't work, push the reset button!*



www.forth.com

Interactive Loops

- A good use of the indefinite loop is to write **interactive loops** that allow a user to repeat certain portions of a program on demand.
- Remember that we need to keep track of how many numbers had been entered? Let's use another accumulator, called `count`.
- At each iteration of the loop, ask the user if there is more data to process. We need to preset it to “yes” to go through the loop the first time.

Interactive Loops

```
#     A program to average a set of numbers
#     Illustrates interactive loop with two accumulators

def main():
    moredata = "yes"
    sum = 0.0
    count = 0
    while moredata[0].lower() == 'y':
        x = float(input("Enter a number >> "))
        sum += x
        count += 1
    moredata = input("Do you have more numbers (yes or no)? ")
print("\nThe average of the numbers is", sum / count)
```

- Using string indexing (`moredata[0]`) allows us to accept “y”, “yes”, “Y” to continue the loop

Interactive Loops

Enter a number >> 32

Do you have more numbers (yes or no)? y

Enter a number >> 45

Do you have more numbers (yes or no)? yes

Enter a number >> 34

Do you have more numbers (yes or no)? yup

Enter a number >> 76

Do you have more numbers (yes or no)? y

Enter a number >> 45

Do you have more numbers (yes or no)? nah

The average of the numbers is 46.4

Sentinel Loops

- A **sentinel loop** continues to process data until reaching a special value that signals the end.
- This special value is called the **sentinel**.
- The sentinel must be distinguishable from the data since it is not processed as part of the data.

Sentinel Loops

```
get the first data item  
while item is not the sentinel  
    process the item  
    get the next data item
```

- The first item is retrieved before the loop starts. This is sometimes called the **priming read**, since it gets the process started.
- If the first item is the sentinel, the loop terminates and no data is processed.
- Otherwise, the item is processed and the next one is read.
- Assume we are averaging test scores. We can assume that there will be no score below 0, so a negative number will be the sentinel.

Sentinel Loops

```
#      A program to average a set of numbers
#      Illustrates sentinel loop using negative input as sentinel

def main():
    sum = 0.0
    count = 0
    x = float(input("Enter a number (negative to quit) >> "))
    while x >= 0:
        sum += x
        count += 1
        x = float(input("Enter a number (negative to quit) >> "))
    print("\nThe average of the numbers is", sum / count)
```

Sentinel Loops

Enter a number (negative to quit) >> 32

Enter a number (negative to quit) >> 45

Enter a number (negative to quit) >> 34

Enter a number (negative to quit) >> 76

Enter a number (negative to quit) >> 45

Enter a number (negative to quit) >> -1

The average of the numbers is 46.4

Sentinel Loops

- Now we can use of the interactive loop without the hassle of typing 'y' all the time.
- BUT we can't average a set of positive **and negative** numbers.
- If we do this, our sentinel can no longer be a number.
- We could input all the information as strings.
- Valid input would be converted into numeric form.
Use a character-based sentinel.
- We could use the *empty string* ("")!

Sentinel Loops

initialize sum to 0.0

initialize count to 0

input data item as a string xStr

while xStr is not empty

 convert xStr to a number x

 add x to sum

 add 1 to count

 input next data item as a string xStr

Output sum / count

Sentinel Loops

```
# A program to average a set of numbers
# Using empty string as loop sentinel

def main():
    sum = 0.0
    count = 0
    xStr = input("Enter a number (<Enter> to quit) >> ")
    while xStr != "":
        sum += float(xStr)
        count += 1
        xStr = input("Enter a number (<Enter> to quit) >> ")
    print("\nThe average of the numbers is", sum / count)
```

Sentinel Loops

```
Enter a number (<Enter> to quit) >> 34
```

```
Enter a number (<Enter> to quit) >> 23
```

```
Enter a number (<Enter> to quit) >> 0
```

```
Enter a number (<Enter> to quit) >> -25
```

```
Enter a number (<Enter> to quit) >> -34.4
```

```
Enter a number (<Enter> to quit) >> 22.7
```

```
Enter a number (<Enter> to quit) >>
```

The average of the numbers is 3.38333333333

Nested Loops

- In the same way that you have an if statement within an if statement, you can have loops within loops
- For example, rather than having 1 number per input line, have multiple, comma-separated numbers per line

Nested Loops

```
# average7.py
#     Computes the average of numbers listed in a file.
#     Works with multiple numbers on a line.

def main():
    fileName = input("What file are the numbers in? ")
    infile = open(fileName, 'r')
    sum = 0.0
    count = 0
    for line in infile:
        # update sum and count for values in line
        for xStr in line.split(","):
            sum += float(xStr)
            count += 1
    print("\nThe average of the numbers is", sum / count)
```

Nested Loops

- The loop that processes the numbers in each line is indented inside of the file processing loop.
- The outer while loop iterates once for each line of the file.
- For each iteration of the outer loop, the inner for loop iterates as many times as there are numbers on the line.
- When the inner loop finishes, the next line of the file is read, and this process begins again.

Nested Loops

- Designing nested loops –
 - *Design the outer loop without worrying about what goes inside*
 - *Design what goes inside, ignoring the outer loop.*
 - *Put the pieces together, preserving the nesting.*

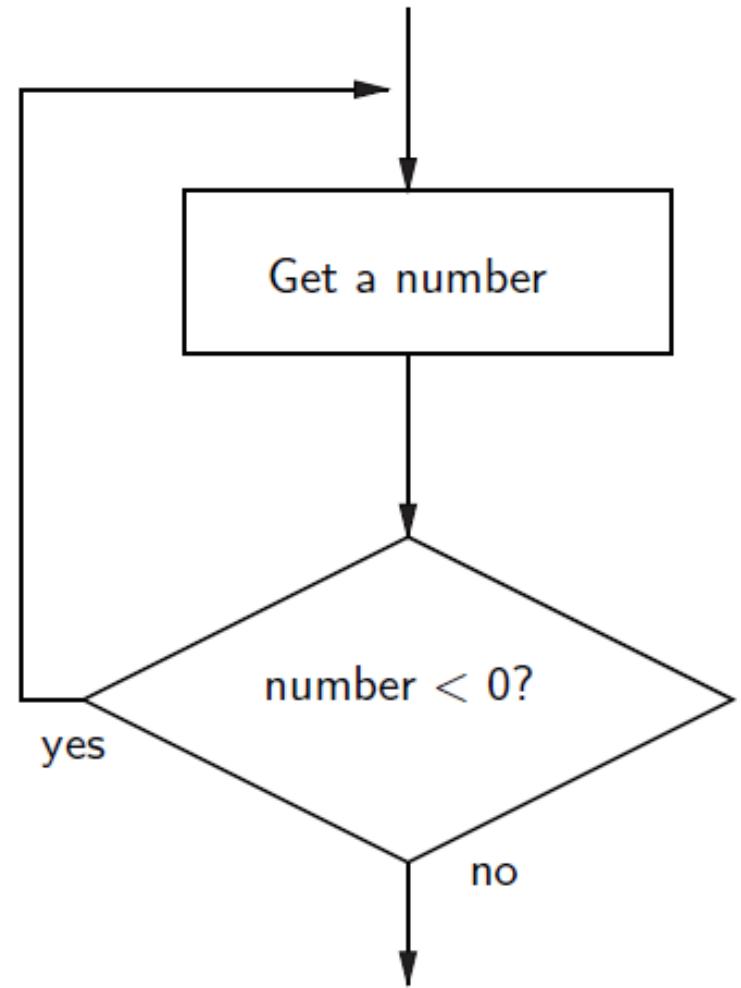
Other Loop Structures – Post-Test Loop

- Say we want to write a program that is supposed to get a nonnegative number from the user.
- If the user types an incorrect input, the program asks for another value.
- This process continues until a valid value has been entered.
- This process is *input validation*.

Post-Test Loop

repeat

 get a number from the user
 until number is ≥ 0



Post-Test Loop

- When the condition test comes after the body of the loop it's called a *post-test loop*.
- A post-test loop always executes the body of the code at least once.
- Python doesn't have a built-in statement to do this, but we can do it with a slightly modified while loop.

Post-Test Loop

```
# A program to average a set of numbers
# Using Post-Test loop which will be execute at least once

def main():
    sum = 0.0
    count = 0
    xStr = " "
    while xStr != "":
        xStr = input("Enter a number (<Enter> to quit) >> ")
        sum += float(xStr)
        count += 1
    print("\nThe average of the numbers is", sum / count)
```

Post-Test Loop

- Some programmers prefer to simulate a post-test loop by using the Python `break` statement.
- Executing `break` causes Python to immediately exit the enclosing loop.
- `break` is sometimes used to exit what looks like an infinite loop.

Post-Test Loop

- The same algorithm implemented with a break:

```
while True:  
    xStr = input("Enter a number (<Enter> to quit) >> ")  
  
    if xStr == "":  
  
        break # Exit loop
```

- A while loop continues as long as the expression evaluates to true. Since True *always* evaluates to true, it looks like an infinite loop!

Loop and a Half

- Stylistically, some programmers prefer the following approach:

```
while True:  
    number = float(input("Enter a positive number: "))  
    if number >= 0: break # if valid number exit loop  
    print("The number you entered was not positive")
```

- Here the loop exit is in the middle of the loop body. This is what we mean by a *loop and a half*.

Loop and a Half

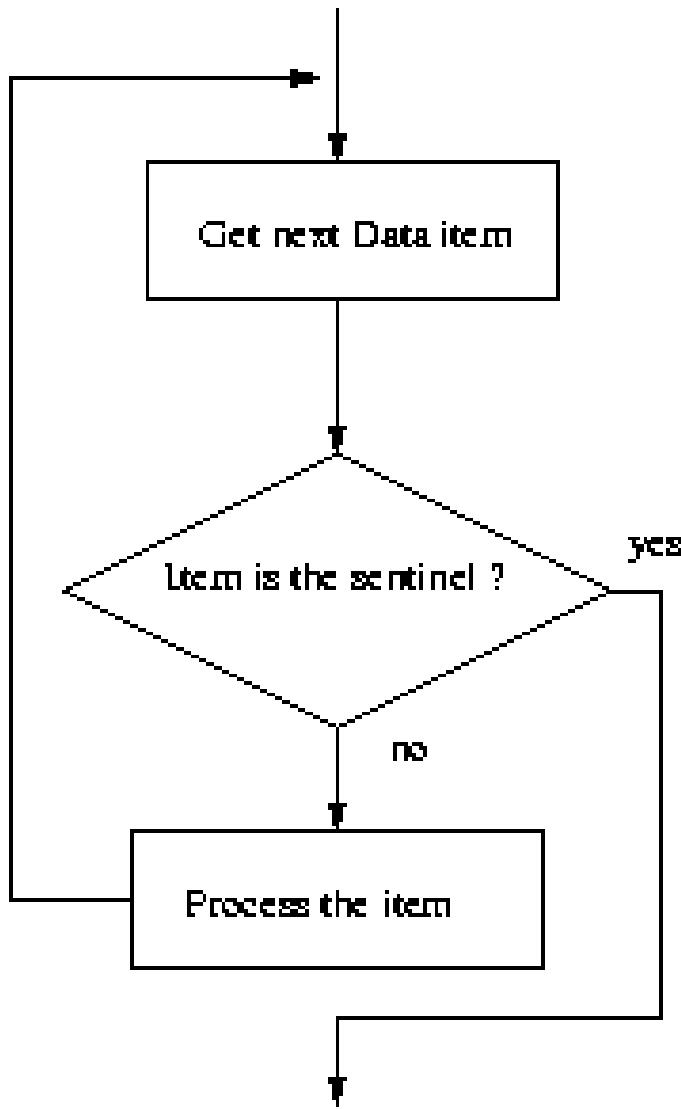
- The loop and a half is an elegant way to avoid the priming read in a sentinel loop.

while True:

```
# get next data item  
# if the item is the sentinel: break  
# process the item
```

- This method is faithful to the idea of the sentinel loop, the sentinel value is not processed!

Loop and a Half



Loop and a Half

- To use or not use break. That is the question!
- The use of break is mostly a matter of style and taste.
- Avoid using break often within loops, because the logic of a loop is hard to follow when there are multiple exits.

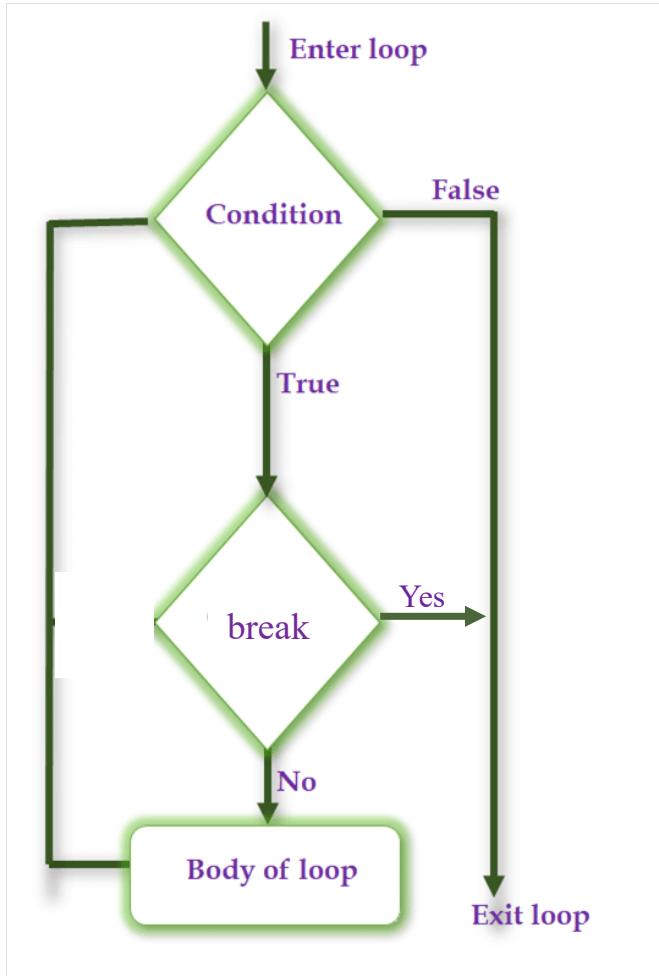
continue statement

- Continue statement returns the control to the beginning of the loop

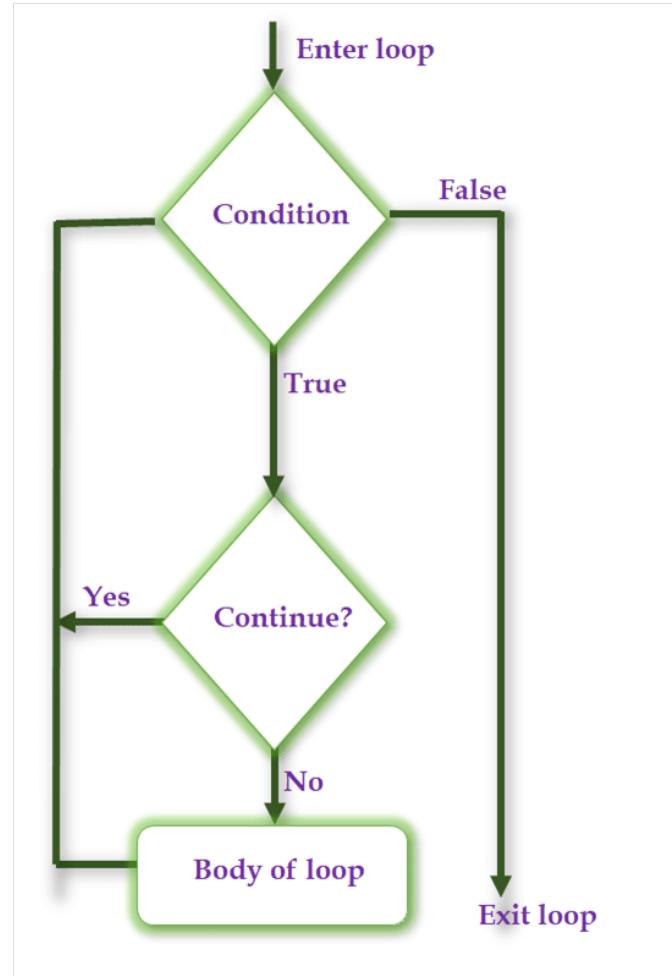
```
# print only even numbers up to 10
for i in range(11):
    if i % 2 == 1: # % is "modulus" operator
        continue
    print(i)
```

break and continue comparison

break statement



continue statement



Summary

- Indefinite loops
- Interactive loops
- Sentinel loops
- Post-Test loops
- **Break** statement
- Loop and a Half
- **Continue** statement



Lecture 13

Loop Examples

Objectives

- Loop revision
- Break statement
- Continue statement
- Loop examples

for Loop: Revision

```
for i in range(10):  
    # do something  
#-----  
myList = [2,3,4,9,10]  
for x in myList:  
    # iterates through the list elements  
    # do something that involves the list elements  
#-----  
myString = "hello there, hello world!"  
for ch in myString:  
    # iterates through the string characters  
#-----  
infile = open(someFile, "r")  
for line in infile:  
    # iterate through the lines of the file  
infile.close()
```

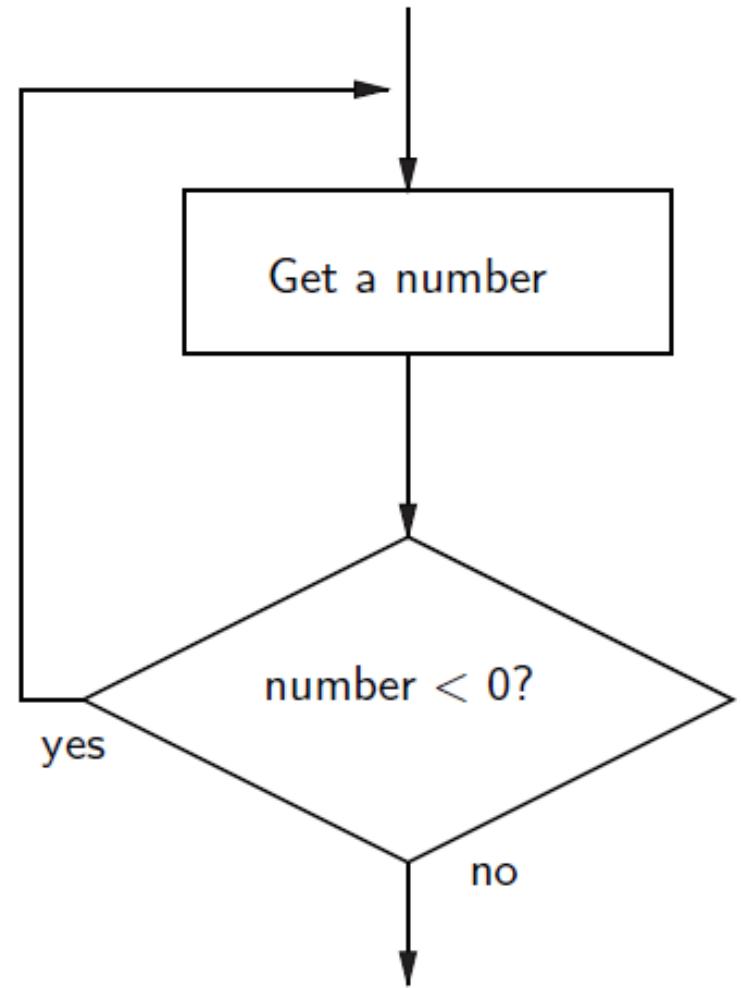
while loop: revision

```
while <condition>:  
    # do something  
#-----  
# program to list first 10 numbers  
# valid but poor use of while  
  
i = 0  
while i < 10:  
    print(i)  
    i += 1  
#-----  
# program to guess a secret number  
n = 7 # secret number  
guess = 1  
while guess != n:  
    guess = int(input("Please guess a number between 0 and 10"))  
print("You guessed correctly")
```

Post-Test Loop

repeat

 get a number from the user
 until number is ≥ 0



Nested Loops

- Designing nested loops –
 - *Design the outer loop without worrying about what goes inside*
 - *Design what goes inside, ignoring the outer loop.*
 - *Put the pieces together, preserving the nesting.*

Loop and a Half

- Stylistically, some programmers prefer the following approach:

```
while True:  
    number = float(input("Enter a positive number: "))  
    if number >= 0: break # if valid number exit loop  
    print("The number you entered was not positive")
```

- Here the loop exit is in the middle of the loop body. This is what we mean by a *loop and a half*.

Loop and a Half

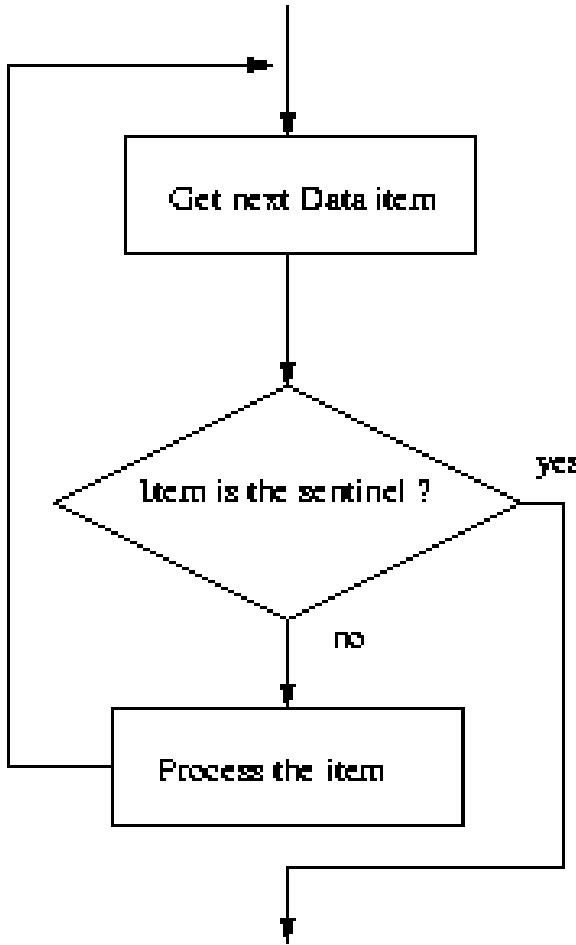
- The loop and a half is an elegant way to avoid the priming read in a sentinel loop.

while True:

```
# get next data item  
# if the item is the sentinel: break  
# process the item
```

- This method is faithful to the idea of the sentinel loop, the sentinel value is not processed!

Loop and a Half



Loop and a Half

- To use or not use break. That is the question!
- The use of break is mostly a matter of style and taste.
- Avoid using break often within loops, because the logic of a loop is hard to follow when there are multiple exits.

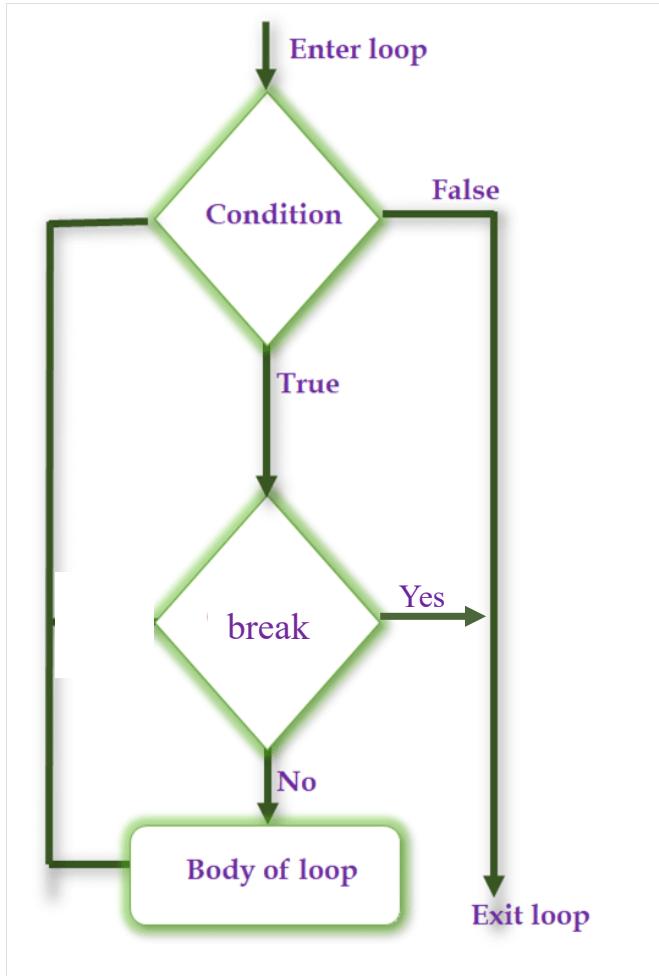
continue statement

- Continue statement returns the control to the beginning of the loop

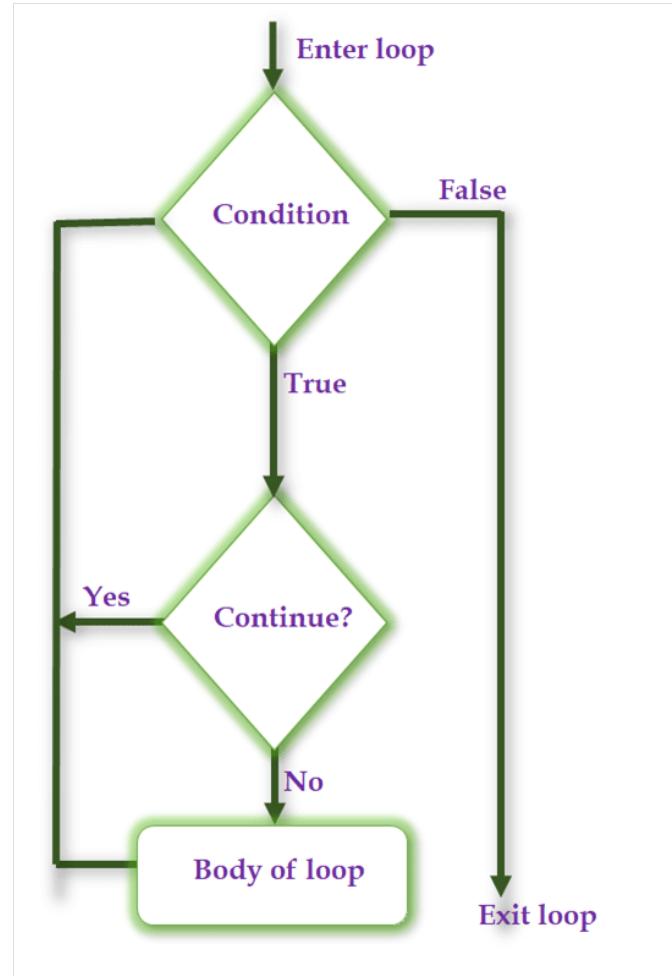
```
# print only even numbers up to 10
for i in range(11):
    if i % 2 == 1: # % is "modulus" operator
        continue
    print(i)
```

break and continue comparison

break statement



continue statement



Loop Example: Prime number

- Find whether a number is prime or not
- Find list of prime number up to N
- Find N prime numbers
- Find N prime numbers using break

Finding whether a number is prime or not ?

```
def primestatus(N):  
    if N < 2:  
        return False  
    elif N < 4:  
        return True  
    else:  
        for i in range(2,N//2+1):  
            if N % i == 0:  
                return False  
    return True
```

Find list of prime number up to N

```
def primelist(N):  
    if N < 2:  
        return []  
    elif N == 2:  
        return [2]  
    else:  
        plist = [2,3]  
        status = True  
        for num in range(4,N+1):  
            for i in range(2,num//2 + 1):  
                if num % i == 0:  
                    status = False  
            if status:  
                plist.append(num)  
            status = True  
    return plist
```

Find N prime numbers

```
# Find the first N prime numbers
# Author: Michael J Wise
def primes(N) :
    primelist = [2,3]
    for pno in range(2,N) :
        i = primelist[-1] + 2 # start search for next one where
        primefound = False      # where last one left off
        while not primefound : # Test successive odd numbers
            factorfound = False
            for divisor in primelist : #Only use previous primes
                if i % divisor == 0 :
                    factorfound = True
                if factorfound : # not prime
                    i += 2
                else :
                    primelist.append(i)
                    primefound = True
    return(primelist)
```

Finding N prime numbers – with break

```
# Find the first N prime numbers (further optimised)
import math
def primes(N) :
    primelist = [2,3]
    for pno in range(2,N) :
        i = primelist[-1] + 2 # start search for next one where left off
        while True :
            factorfound = False
            if N > 100 : # time for sqrt not worth it for N<=100
                stopat = int(math.sqrt(i))
            for divisor in primelist : # Only test previous primes
                if N > 100 and divisor > stopat :
                    break # From divisor search loop
                if i % divisor == 0 :
                    factorfound = True
                    break
            if factorfound : # not prime, keep searching
                i += 2
            else :
                primelist.append(i)
                break # Got a prime, break from this prime search
    return(primelist)
```

Summary

- break statement
- continue statement
- Example: prime numbers



THE UNIVERSITY OF
WESTERN
AUSTRALIA

Lecture 14

Exceptions

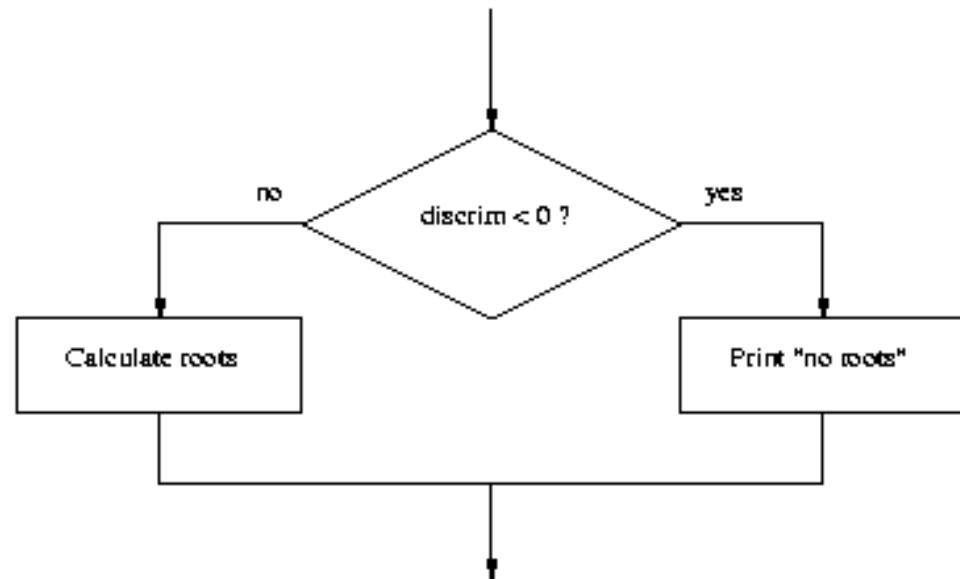
Revision – Simple Decisions

- While sequential flow is a fundamental programming concept, it is not sufficient to solve every problem.
- We need to be able to alter the sequence of a program to suit a particular situation.
- The `if` statement facilitates conditional execution of a code block (if condition evaluates to True)

Revision – Two-Way Decisions

- In Python, a two-way decision can be implemented by attaching an else clause onto an if clause.
- This is called an `if-else` statement:

```
if <condition>:  
    <statements>  
else:  
    <statements>
```



Revision – Multi-Way Decisions

- Multi-way decisions are implemented by the if-elif-else construct.
- Allows multiple alternative conditions to be tried, one after the other, until one succeeds – or else if none do.

```
if <condition1>:  
    <case1 statements>  
elif <condition2>:  
    <case2 statements>  
elif <condition3>:  
    <case3 statements>  
...  
else:  
    <default statements>
```

Exception Handling

- In the quadratic program we used decision structures to avoid taking the square root of a negative number, thus avoiding a run-time error.
 - This is true for many programs: decision structures are used to protect against rare but possible errors.
 - In the quadratic example, we checked the data *before* calling `sqrt`. Sometimes functions will check for errors and return a special value to indicate the operation was unsuccessful.
 - E.g., a different square root operation might return -1 to indicate an error (since square roots are never negative, we know this value will be unique).
-

Exception Handling

```
discRt = otherSqrt(b*b - 4*a*c)
if discRt < 0:
    print("No real roots.")
else:
    ...
```

- Sometimes programs get so many checks for special cases that the algorithm becomes hard to follow.
 - *Common in code that deals with user supplied inputs*
- Programs that detect an error condition **raise an exception**
- Programming language designers have come up with a mechanism called **exception handling** to solve this design problem.

Exception Handling

```
>>> x = int(input("Enter an integer: "))
```

```
Enter an integer: a
```

```
Traceback (most recent call last):
```

```
  File "<pyshell>", line 1, in <module>
```

```
    ValueError: invalid literal for int() with  
    base 10: 'a'
```

- `int()` is expecting digits so raised a `ValueError` exception when a letter was entered.

Exception Handling

```
>>> print(spam)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
NameError: name 'spam' is not defined
```

- In this case spam (without quotes) is treated as a variable, but the variable is not defined
- There is a range of named Exceptions
- <https://docs.python.org/3/library/exceptions.html>

Exception Handling

- The programmer can write code that **catches** and deals with exceptions that arise while the program is running, i.e., “Do these steps, and if any problem crops up, handle it this way.”
- Often, all that can be done is for the program to exit **gracefully** with a more useful error message

Exception Handling

- The `try` statement has the following form:

```
try:  
    <body>  
except <ErrorType>:  
    <handler>
```

- That is, you place the statements that could cause problems within a `try` block
- When Python encounters a `try` statement, it attempts to execute the statements inside the `try` body.
- If exception is raised, execute handler
- If there is no error, control passes to the next statement after the `try .. except`.

Exception Handling

```
# A simple program illustrating chaotic behaviour

def main():
    print("This program illustrates a chaotic function")
    try:
        x = float(input("Enter a number between 0 and 1: "))
    except ValueError:
        print("Non number entered. Cannot proceed")
        return
    for i in range(10):
        x = 3.9 * x * (1 - x)
        print(i, x)

main()
```

Exception Handling

- Can have multiple `except` blocks. Multiple `except` statements act like `elif`. If an error occurs, Python will test each `except` looking for one that matches the type of error.
- A bare `except` acts like an `else` and catches any errors without a specific exception type.
- If there was no bare `except` at the end and none of the `except` clauses match, the program would **still crash** and report an error.

Exception Handling Style

- Exceptions are intended for *exceptional* circumstances
- Exceptions should not be used as a substitute for carefully planned if statements

Exception Handling

```
def main():
    print("This program illustrates a chaotic function")
    try:
        x = float(input("Enter a number between 0 and 1: "))
    except ValueError:
        print("Non number entered. Cannot proceed")
        return
    if x > 0 and x < 1.0 :
        for i in range(10):
            x = 3.9 * x * (1 - x)
            print(i, x)
    else:
        print("number entered was not in range 0 .. 1.0")
main()
```

Exception Handling

- A very common use for exception handling is when opening files, which may fail for a range of reasons, many of which are hard to predict and tedious to test for.
 - *File not present or empty (reading)*
 - *Directory permissions do not allow file creation for writing*
 - *File permissions do not allow reading or appending*

Exception Handling – File Opening

```
# Opens named file with specified mode, returning the file handle
# If that results in an exception, message printed and None returned
def test_fileIO(name, mode) :
    mode_list = ["r", "w", "a"]
    if mode not in mode_list:
        print("Unknown file access mode", mode)
        return(None)

    try:
        f = open(name, mode)
    except IOError:
        print("cannot open the file {0:s}".format(name))
        return(None)

    return(f)
```

The real main ()

- We have conventionally called the top level function `main()`
- There is a real main program, and that is the code that exists in the module outside any function/object. It is called `__main__`
- Generally used to allow code to be executed interactively, but not on import

```
if __name__ == "__main__": Two underscores!  
    stuff
```

Summary

- In this lecture we learnt about:
 - *Exceptions as unexpected situations and how to handle them in Python*
 - *Increment assignment*



THE UNIVERSITY OF
WESTERN
AUSTRALIA

Lecture 15

Lists to Dictionaries

Revision - Lists

- String and lists are subclasses of sequence
 - *Lists are **mutable**, but strings are not*
- Items in a list or string are obtained by **indexing**, with list (and string) items numbered from 0
- Lists can contain items of different types, e.g.
`[1, 2.0, "three"]`
- Lists are dynamic (they grow and shrink as required).

Sequence Operations

Operator	Meaning
$\langle \text{seq} \rangle + \langle \text{seq} \rangle$	Concatenation
$\langle \text{seq} \rangle * \langle \text{int-expr} \rangle$	Repetition
$\langle \text{seq} \rangle []$	Indexing
$\text{len}(\langle \text{seq} \rangle)$	Length
$\langle \text{seq} \rangle [:]$	Slicing
$\text{for } \langle \text{var} \rangle \text{ in } \langle \text{seq} \rangle:$	Iteration
$\langle \text{expr} \rangle \text{ in } \langle \text{seq} \rangle$	Membership (Boolean)

Sequence Operations

- Except for the membership check, we've used these operations before on strings.
- The membership operation can be used to see if a certain value appears anywhere in a sequence.

```
>>> lst = [1,2,3,4]
```

```
>>> 3 in lst
```

True

```
>>> month = 1
```

```
>>> year = 2000
```

```
>>> if month in [1,2] : # month == 1 or month == 2  
    year -= 1
```

List Operations

Method	Meaning
<list>.append(x)	Add element x to end of list.
<list>.sort()	Sort the list. A comparison function may be passed as a parameter. By default sorted in ascending order
<list>.reverse()	Reverse the list.
<list>.index(x)	Returns index of first occurrence of x.
<list>.insert(i, x)	Insert x into list at index i.
<list>.count(x)	Returns the number of occurrences of x in list.
<list>.remove(x)	Deletes the first occurrence of x in list.
<list>.pop(i)	Deletes the i^{th} element of the list and returns its value.

List Operations

```
>>> lst = [3, 1, 4, 1, 5, 9]      >>> lst.insert(4, "Hello")
>>> lst.append(2)
>>> lst
[3, 1, 4, 1, 5, 9, 2]
>>> lst.sort()
>>> lst
[1, 1, 2, 3, 4, 5, 9]
>>> lst.reverse()
>>> lst
[9, 5, 4, 3, 2, 1, 1]
>>> lst.index(4)
2
>>> lst.insert(4, "Hello")
>>> lst
[9, 5, 4, 3, 'Hello', 2, 1]
>>> lst.count(1)
2
>>> lst.remove(1)
>>> lst
[9, 5, 4, 3, 'Hello', 2, 1]
>>> lst.pop(3)
3
>>> lst
[9, 5, 4, 'Hello', 2, 1]
```

List Operations

- Most of these methods don't return a value** – they change the contents of the list in some way.

```
>>> lst.sort()
```

```
>>> lst.sort(reverse=True)
```

- 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.

** They return None

List Operations

- ```
>>> myList=[34, 26, 0, 10]
>>> del myList[1]
>>> myList
[34, 0, 10]
>>> del myList[1:3]
>>> myList
[34]
```
- `del` isn't a list method, but a built-in operation that can also be used on list items.

# Tuples

---

- A *tuple* is a sequence which looks like a list but uses () rather than [ ].
- Tuples are sequences that are **immutable**, so are used to represent sequences that are not supposed to change,
  - e.g. *student-mark pairs*
  - [ ('Fred', 55), ('Jemima', 68), ('James', 68) ]
  - *Sorting a list of tuples sorts on first member of each tuple*
  - *Turn a list into a tuple by using the tuple() function*

# Dictionaries

---

- After lists, a **dictionary** is probably the most widely used collection/compound data type.
- Dictionaries are not as common in other languages as lists (arrays).
- Lists are sequential
  - *To find a particular need to search from the start.*
  - *Do you find a book in the library starting from Dewey number (000 is computer science!)*
- Use catalogue!

# Dictionaries

---

- Dictionaries use key-value pairs
- There are lots of examples!
  - *Names and phone numbers*
  - *Usernames and passwords*
  - *State names and capitals*
- A collection that allows us to look up information associated with arbitrary keys is called a **mapping**.
- Python dictionaries are *mappings*. Other languages call them *hashes* or *associative arrays*.

# Dictionaries

---

- Dictionaries can be created in Python by listing key-value pairs inside of curly braces.
- Keys and values are joined by : and are separated with commas.

```
>>>passwd = {"guido": "superprogrammer",
"turing": "genius", "bill": "monopoly"}
```

- We use an indexing notation to do lookups

```
>>> passwd["guido"]
'superprogrammer'
```

- Unlike list indexes, which are integers related to position in the list, dictionary indexes can be almost anything

# Dictionaries

---

- <dictionary>[<key>] returns the object with the associated key.
- Dictionaries are mutable.

```
>>> passwd["bill"] = "bluescreen"
```

```
>>> passwd
```

```
{'guido': 'superprogrammer', 'bill':
'bluescreen', 'turing': 'genius'}
```

- Did you notice the dictionary printed out in a different order than it was created?

# Initialising Dictionaries

---

- Dictionaries can be created directly

```
dayno is in the range 0..6

dow_map = { 0:"Sunday", 1:"Monday", 2:"Tuesday",
 3:"Wednesday", 4:"Thursday", 5:"Friday",
 6:"Saturday" }

daystr = dow_map[dayno]

print(daystr)
```

# Initialising Dictionaries

---

- Dictionaries can also be created incrementally. That is, start with an empty dictionary and add the key-value pairs one at a time.
- For example, assume the file `passwords` contains comma-separated pairs of user IDs and passwords

```
passwd_dir = {}

for line in open('passwords', 'r'):
 user, pw = line.strip().split(',')
 passwd_dir[user] = pw
```

---

# Dictionary Operations

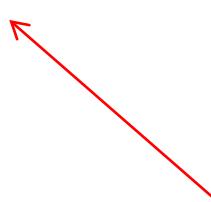
---

| Method                       | Meaning                                                                                     |
|------------------------------|---------------------------------------------------------------------------------------------|
| <key> in <dict>              | Returns true if dictionary contains the specified key, false if it doesn't.                 |
| <dict>.keys()                | Returns a sequence of keys.                                                                 |
| <dict>.values()              | Returns a sequence of values.                                                               |
| <dict>.items()               | Returns a sequence of tuples (key, value) representing the key-value pairs (i.e. 2-tuples). |
| del <dict>[<key>]            | Deletes the specified entry.                                                                |
| <dict>.clear()               | Deletes all entries.                                                                        |
| for <var> in <dict>:         | Loop over the keys.                                                                         |
| <dict>.get(<key>, <default>) | If dictionary has key, returns its value; otherwise returns default.                        |
| <dict>[<key>]                | If dictionary has key, return its value; otherwise exception raised                         |

# Dictionary Operations

---

```
>>> list(passwd.keys())
['guido', 'turing', 'bill']
>>> list(passwd.values())
['superprogrammer', 'genius', 'bluescreen']
>>> list(passwd.items())
[('guido', 'superprogrammer'), ('turing', 'genius'),
 ('bill', 'bluescreen')]
>>> "bill" in passwd
True
>>> "fred" in passwd
False
```



List of 2-tuples

# Dictionary Operations

---

```
>>> passwd.get("guido", "unknown")
'superprogrammer'
>>> passwd.get("fred", "unknown")
'unknown'
>>> passwd["fred"]
Traceback (most recent call last):
 File "<pyshell>", line 1, in <module>
 KeyError: 'fred'
>>> passwd.clear()
>>> passwd
{ }
```

# Example Program: Word Frequency

---

- We want to write a program that analyzes text documents and counts how many times each word appears in the document.
- This kind of analysis is sometimes used as a crude measure of the style similarity between two documents and is used by automatic indexing and archiving programs (like Internet search engines).

# Example Program: Word Frequency

---

- This is a multi-accumulator problem!
- We need a count for each word that appears in the document.
- We can use a loop that iterates over each word in the document, incrementing the appropriate accumulator.
- The catch: we will likely need hundreds, perhaps thousands of these accumulators!

# Example Program: Word Frequency

---

- Let's use a dictionary where strings representing the words are the keys and the values are ints that count up how many times each word appears.
  - *The mapping is: <string> <int>*
- To update the count for a particular word,  $w$ , we need something like:

```
counts[w] += 1
```
- One problem – the first time we encounter a word it will not yet be in `counts`.

# Example Program: Word Frequency

---

- Attempting to access a nonexistent key produces a run-time KeyError.

Pseudo-code

if w is already in counts:

    add one to the count for w

else:

    set count for w to 1

- How can this be implemented?

# Example Program: Word Frequency

---

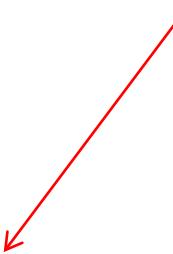
```
if w in counts:
 counts[w] += 1
else:
 counts[w] = 1
```

Can't do this in Python 2

- A more elegant approach:

```
counts[w] = counts.get(w, 0) + 1
```

- If  $w$  is not already in the dictionary, this `get` will return 0, and the result is that the entry for  $w$  is set to 1.



# Example Program: Word Frequency

---

- The other tasks include
  - *Convert the text to lowercase (so occurrences of “Python” match “python”)*
  - *Eliminate punctuation (so “python!” matches “python”)*
  - *Split the text document into a sequence of words*

# Example Program: Word Frequency

---

```
get the sequence of words from the file
fname = input("File to analyze: ")
try:
 text = open(fname, 'r').read()
except IOError:
 print("Cannot open {}".format(fname))
 return
text = text.lower()
for ch in '!#$%&()*+,-./:;<=>?@[\\"^_`{|}~':
 text = text.replace(ch, ' ')

```

# Example Program: Word Frequency

---

- Variable `text` has all the words in the file. Multiple spaces not a problem for `split()`

```
words = text.split()
```

- Loop through the words to build the counts dictionary

```
counts = {}

for w in words:

 counts[w] = counts.get(w, 0) + 1
```

# Example Program: Word Frequency

---

- print a list of words in alphabetical order with their associated counts

```
get list of words that appear in document
each word (i.e. key) is found only once!
uniqueWords = list(counts.keys())
```

```
put list of words in alphabetical order
uniqueWords.sort()
```

```
print words and associated counts
for w in uniqueWords:
 print(w, counts[w])
```

---

# Example Program: Word Frequency

---

- This will probably not be very useful for large documents with many words that appear only a few times.
  - *Result will be a huge list*
- A more interesting analysis is to print out the counts for the  $n$  most frequent words in the document.
- To do this, we'll need to create a list that is sorted by counts (most to fewest), and then select the first  $n$  items.

# Example Program: Word Frequency

---

- We can start by getting a list of key-value pairs using the `items` method for dictionaries.

```
pairs = list(count.items())
```

- `pairs` will be a list of tuples like

```
[('foo', 5), ('bar', 7), ('spam', 376)]
```

- If we try to sort them with `pairs.sort()`, they will be in ascending order of first component, i.e. dictionary order of the words.

```
[('bar', 7), ('foo', 5), ('spam', 376)]
```

# Example Program: Word Frequency

---

- Not what we wanted.
- To sort the items by frequency, we need a function that will take a tuple (here, 2-tuple) and return the second term, i.e. count.

```
def byCount(pair):
 return pair[1]
```

- To sort the list by frequency:

```
pairs.sort(key=byCount)
```

# Example Program: Word Frequency

---

- We're getting there!
- What if have multiple words with the same number of occurrences? We'd like them to print in alphabetical order.
- That is, we want the list of pairs primarily sorted by count, but sorted alphabetically within each level.

# Example Program: Word Frequency

---

- Looking at the documentation for `sort`, it says this method performs a “*stable* sort *in place*”.
  - “*In place*” means *the method modifies the list that it is applied to, rather than producing a new list.*
  - *Stable* means equivalent items (*equal keys*) stay *in the same relative position to each other as they were in the original list.*

# Example Program: Word Frequency

---

- If all the words were in alphabetical order before sorting them by frequency, words with the same frequency will be in alphabetical order!
- We just need to sort the list twice – first by words, then by frequency.

```
pairs.sort() # orders pairs alphabetically
pairs.sort(key = byCount, reverse = True) # orders by count
```

- Setting `reverse` to `True` tells Python to sort the list in reverse order.

# Example Program: Word Frequency

---

- Now we are ready to print a report of the  $n$  most frequent words.
- Here, the loop index  $i$  is used to get the next pair from the list of items.
- That pair is unpacked into its word and count components.
- The word is then printed left-justified in fifteen spaces, followed by the count right-justified in five spaces.

# Example Program: Word Frequency

---

```
for i in range(n) :
 word, count = pairs[i]
 print ("{0:<15} {1:>5}".format(word, count))
```

# Example Program: Word Frequency

---

```
A program to count word frequencies in text file
def byCount(pair): # service function, select second of pair
 return pair[1]

def main():
 print("This program counts word frequency in a file and")
 print("prints a report on the n most frequent words.\n")
 # get the sequence of words from the file
 fname = input("File to analyze: ")
 text = open(fname, 'r').read()
 text = text.lower()
 for ch in '!"#$%&()*+,-./:;=>?@[\\"`^_{|}~':
 text = text.replace(ch, ' ')

```

# Example Program: Word Frequency

---

```
words = text.split()

construct a dictionary of word counts
counts = { }
for w in words:
 counts[w] = counts.get(w, 0) + 1
output analysis of n most frequent words.
n = int(input("Output analysis of how many words? "))
items = list(counts.items()) # word-count pair list
items.sort() # alphabetic sort
items.sort(key=byCount, reverse=True)
for i in range(n):
 word, count = items[i]
 print("{0:<15}{1:>5}".format(word, count))
```

# Summary

---

- We completed looking at Python lists, noting that many of the functions are actually methods that change the input list, esp. append and sort.
- We looked at tuples, as a special sort of list.
- We looked at dictionaries, as a mapping from keys to values which is not restricted to the order of items



THE UNIVERSITY OF  
**WESTERN**  
**AUSTRALIA**

---

# Lecture 16

## Simulation and Design

---

# Objectives

---

- To understand the potential applications of simulation as a way to solve real-world problems.
- To understand pseudorandom numbers and their application in Monte Carlo simulations.
- To understand and be able to apply top-down and spiral design techniques in writing complex programs.
- To understand unit-testing and be able to apply this technique in the implementation and debugging of complex programming.

# Simulation

---

- *Simulation* can solve real-world problems by modeling real-world processes to provide otherwise unobtainable information.
  - *Physical techniques unavailable or very expensive*
- Computer simulation is used to predict the weather, design aircraft, create special effects for movies, etc.

# Simulating Racquetball Games

---

- Denny often plays racquetball with players who are slightly better than he is.
- Denny usually loses his matches!
- Shouldn't players who are *a little* better win *a little* more often?
- Susan suggests that they write a simulation to see if slight differences in ability can cause such large differences in scores.

# Background

---

- Racquetball is played between two players using a racquet to hit a ball in a four-walled court.
  - *Similar to squash*
- One player starts the game by putting the ball in motion – *serving*.
- Players try to alternate hitting the ball to keep it in play, referred to as a *rally*. The rally ends when one player fails to hit a legal shot.

# Background

---

- The player who misses the shot loses the rally. If the loser is the player who served, serving passes to the other player.
- If the server wins the rally, a point is awarded and keeps serving. Players can only score points when they are serving.
- The first player to reach 15 points wins the game.

# Analysis and Specification

---

- In our simulation, the ability level of the players will be represented by the probability that the player wins the rally when he or she serves.
- Example: Players with a 0.60 probability win a point on 60% of their serves.
- The program will prompt the user to enter the service probability for both players and then simulate multiple games of racquetball.
- The program will then print a summary of the results.

# Analysis and Specification

---

- Input: The program prompts for and gets the service probabilities of players A and B. The program then prompts for and gets the number of games to be simulated.

What is the probability player A wins a serve?

What is the probability player B wins a serve?

How many games to simulate?

- Output: The program then prints out a nicely formatted report showing the number of games simulated and the number of wins and the winning percentage for each player.

Games simulated: 500

Wins for A: 268 (53.6%)

Wins for B: 232 (46.4%)

- Assumption: Assume that all inputs are valid

# PseudoRandom Numbers

---

- When we say that player A wins 50% of the time, that doesn't mean they win every other game. Rather, it's more like a coin toss.
- Overall, half the time the coin will come up heads, the other half the time it will come up tails, but one coin toss does not effect the next (it's possible to get 5 heads in a row, with probability  $1/32$ ).
- Many simulations require events to occur with a certain likelihood. These sorts of simulations are called Monte Carlo simulations because the results depend on “chance” probabilities.
- *Nothing random about computers but need to have a source of “random” numbers*

# PseudoRandom Numbers

---

- A pseudorandom number generator works by starting with a seed value – current date and time. This value is given to a function to produce a “random” number.
- The next time a random number is required, the current value is fed back into the function to produce a new number.
- This sequence of numbers appears to be random, but if you start the process over again with the same seed number, you’ll get the same sequence of “random” numbers.
- Python provides the random library module that contains a number of functions for working with pseudorandom numbers

# PseudoRandom Numbers

---

- These functions derive an initial seed value from the computer's date and time when the module is loaded, so each time a program is run a different sequence of random numbers is produced.
- The four functions of greatest interest are `randrange`, `choice`, `random` and `seed`.
  - `randrange ()` – *randomly select an integer value from a range. range () rules apply*
    - E.g. `randrange (1, 6)` returns some number from `[1, 2, 3, 4, 5]` and `randrange (5, 105, 5)` returns a random multiple of 5 between 5 and 100, inclusive. (End value not included, as usual)

# PseudoRandom Numbers

---

- *choice () – Choose a random member of a list*
    - E.g. `choice( [ 'A' , 'B' ] )`
  - *random () – Returns a random number in the range [0 .. 1.0) (0 can be returned, but not 1.0*
    - ```
>>> import random as rnd
>>> rnd.random()
0.79432800912898816
>>> rnd.random()
0.00049858619405451776
```
 - *seed () – Assign the random number generator a fixed starting point (to give reproducible behaviour during testing)*
-

PseudoRandom Numbers

- The racquetball simulation makes use of the `random` function to determine if a player has won a serve.
- Suppose a player's probability when they serve is 70%, or 0.70.

```
if <player wins serve>:
```

```
    score = score + 1
```

- We need to insert a probabilistic function that will succeed 70% of the time.

PseudoRandom Numbers

- The racquetball simulation makes use of the `random` function to determine if a player has won a serve.
- Suppose we generate a random number between 0 and 1. Exactly 70% of the interval 0..1 is to the left of 0.7.
- So 70% of the time the random number will be < 0.7 , and it will be ≥ 0.7 the other 30% of the time.
 - *The = goes on the lower end since the random number generator can produce a 0 but not a 1.*
- If `prob` represents the probability of winning the serve, the condition `random.random() < prob` will succeed with the correct probability.

```
if random.random() < prob:  
    score += 1
```

Top-Down Design

- In **top-down design**, a complex problem is expressed as a solution in terms of smaller, simpler problems.
- These smaller problems are then solved by expressing them in terms of smaller, simpler problems.
- This continues until the problems are trivial to solve. The smaller pieces are then put back together as a solution to the original problem!

Top-Down Design

- Typically a program uses the *input, process, output* pattern.
- The algorithm for the racquetball simulation:

Print an introduction

Get the inputs: probA, probB, n

Simulate n games of racquetball using probA and probB

Print a report on the wins for playerA and playerB

- Whatever we don't know how to do, we'll ignore for now.
- Assume that all the components needed to implement the algorithm have been written already, and that your task is to finish this top-level algorithm using those components.

Top-Down Design

- First we print an introduction.
- This is easy, and we don't want to bother with it.

```
def main():  
    printIntro()
```

- We assume that there's a `printIntro()` function that prints the instructions!

Top-Down Design

- The next step is to get the inputs.
- We know how to do that! Let's assume there's already a component that can do that called `getInputs()`.
- `getInputs()` gets the values for `probA`, `probB`, and `n`.

```
def main():
    printIntro()
    probA, probB, n = getInputs()
```

Top-Down Design

- If you were going to simulate the game by hand, what inputs would you need?
 - $probA$
 - $probB$
 - n
 - What values would you need to get back?
 - *The number of games won by player A*
 - *The number of games won by player B*
 - These must be the outputs from the `simNGames` function.
-

Top-Down Design

- We now know that the main program must look like this:

```
def main():  
    printIntro()  
    probA, probB, n = getInputs()  
    winsA, winsB = simNGames(n, probA, probB)
```

- What information would you need to be able to produce the output from the program?
- You need to know how many wins there were for each player – these will be the inputs to the next function.

Top-Down Design

- The complete main program:

```
def main():
    printIntro()
    probA, probB, n = getInputs()
    winsA, winsB = simNGames(n, probA, probB)
    printSummary(winsA, winsB)
```

- The name, parameters, and expected return values of these functions have been specified. This information is known as the **interface (API)** or **signature** of the function.
 - `main()` *in the programming projects was an API*

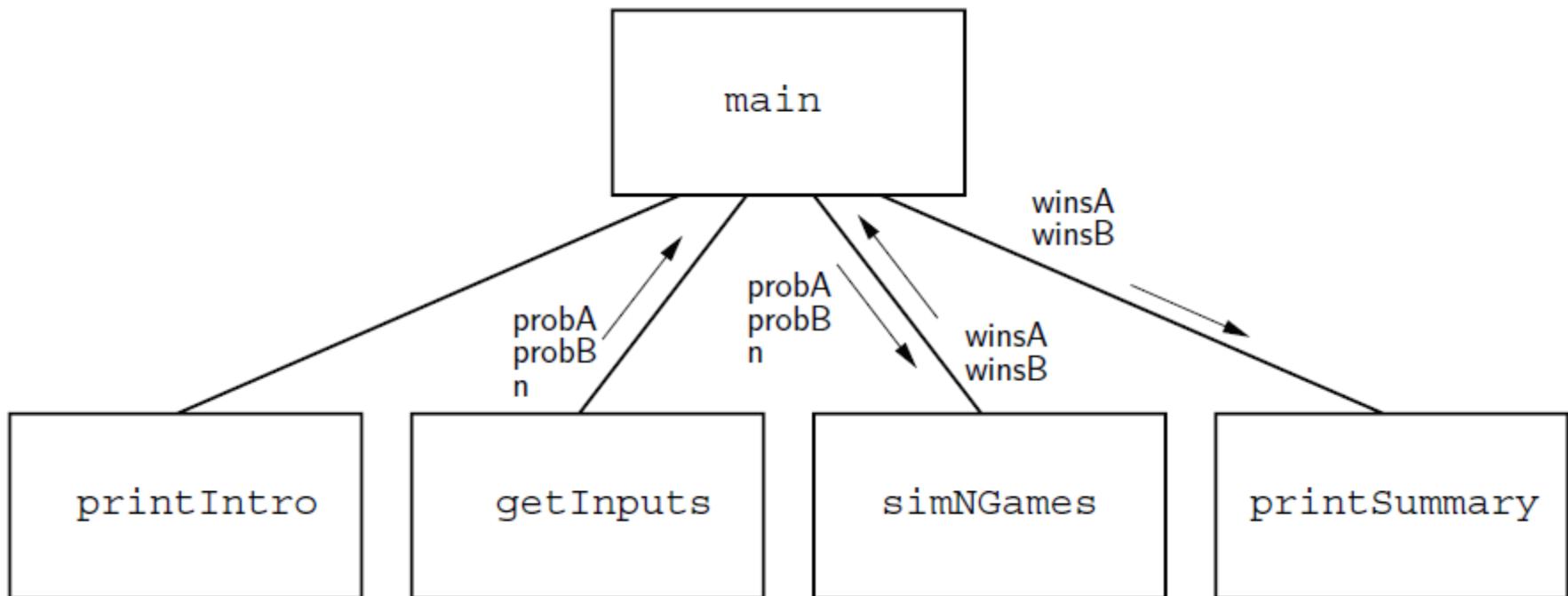
Separation of Concerns

- Having this information (the *signatures*), allows us to work on each of these pieces independently.
- For example, as far as `main()` is concerned, *how* `simNGames` works is not a concern as long as passing the number of games and player probabilities to `simNGames` causes it to return the correct number of wins for each player.

Separation of Concerns

- In a **structure chart** (or **module hierarchy**), each component in the design is a rectangle.
- A line connecting two rectangles indicates that the one above uses the one below.
- The arrows and annotations show the interfaces between the components.

Separation of Concerns



Separation of Concerns

- At each level of design, the interface tells us which details of the lower level are important.
- The general process of determining the important characteristics of something and ignoring other details is called **abstraction**.
- The top-down design process is a systematic method for discovering useful abstractions.

Second Level Design

- The next step is to repeat the process for each of the modules defined in the previous step!
- The `printIntro` function should print an introduction to the program. The code for this is straightforward.

```
def printIntro():  
    # Prints an introduction to the program  
  
    print("This program simulates a game of racquetball between two")  
    print('players called "A" and "B". The ability of each player')  
    print("is indicated by a probability (a number between 0 and 1)")  
    print("that the player wins the point when serving. Player A")  
    print("always has the first serve.\n")
```

- Since there are no new functions, there are no changes to the structure chart.
-

Second Level Design

- In `getInputs`, we prompt for and get three values, which are returned to the main program.

```
def getInputs():  
    # RETURNS probA, probB, number of games to simulate  
    a = float(input("What is the prob. player A wins a serve? "))  
    b = float(input("What is the prob. player B wins a serve? "))  
    n = int(input("How many games to simulate? "))  
    return a, b, n
```

Designing simNGames

- This function simulates n games and keeps track of how many wins there are for each player.
- “Simulate n games” sounds like a counted loop, and tracking wins sounds like a good job for accumulator variables.

Initialize winsA and winsB to 0

loop n times

 simulate a game

 if playerA wins

 Add one to winsA

 else

 Add one to winsB

Designing simNGames

We already have the function signature:

```
def simNGames(n, probA, probB):  
    # Simulates n games of racquetball between players A, B  
    # RETURNS number of wins for A, number of wins for B
```

With this information, it's easy to get started!

```
def simNGames(n, probA, probB):  
    # Simulates n games of racquetball between players A, B  
    # RETURNS number of wins for A, number of wins for B  
    winsA = 0  
    winsB = 0  
    for i in range(n):
```

Designing simNGames

- The next thing we need to do is simulate a game of racquetball. We're not sure how to do that, so let's put it off until later!
- Let's assume there's a function called `simOneGame` that can do it.
- The inputs to `simOneGame` are easy – the probabilities for each player. But what is the output?

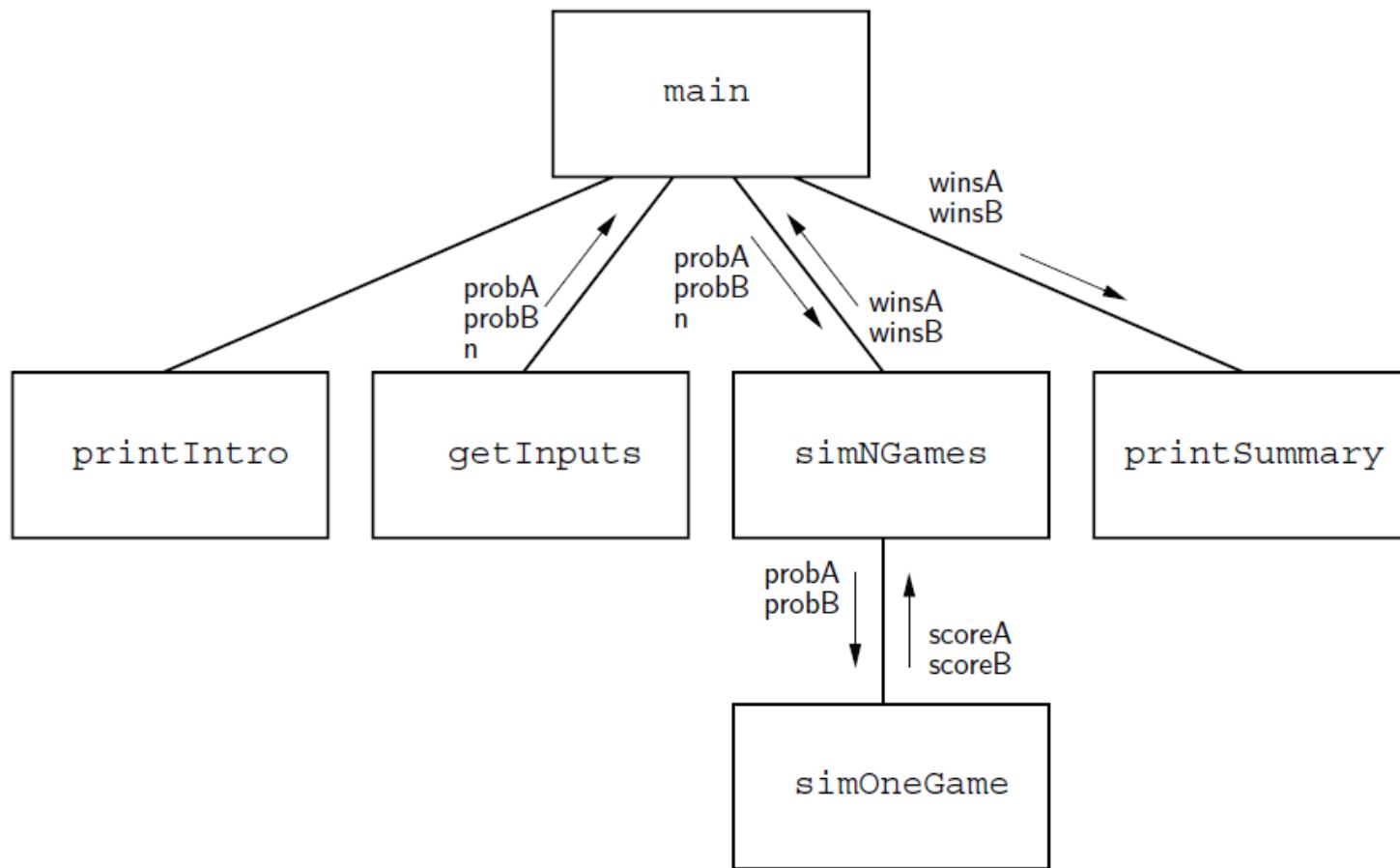
Designing simNGames

- We need to know who won the game. How can we get this information?
- The easiest way is to pass back the final score.
- The player with the higher score wins and gets their accumulator incremented by one.

Designing simNGames

```
def simNGames(n, probA, probB):  
    # Simulates n games of racquetball between players A, B  
    # RETURNS number of wins for A, number of wins for B  
    winsA = winsB = 0  
    for i in range(n):  
        scoreA, scoreB = simOneGame(probA, probB)  
        if scoreA > scoreB:  
            winsA += 1  
        else:  
            winsB += 1  
    return winsA, winsB
```

Designing simNGames



Third-Level Design

- The next function we need to write is `simOneGame`, where the logic of the racquetball rules lies.
- Players keep doing rallies until the game is over, which implies the use of an indefinite loop, since we don't know ahead of time how many rallies there will be before the game is over.
- We also need to keep track of the score and who's serving. The score will be two accumulators, so how do we keep track of who's serving?
- One approach is to use a string value that alternates between “A” or “B”.

Third-Level Design

Initialize scores to 0

Set serving to "A"

Loop while game is not over:

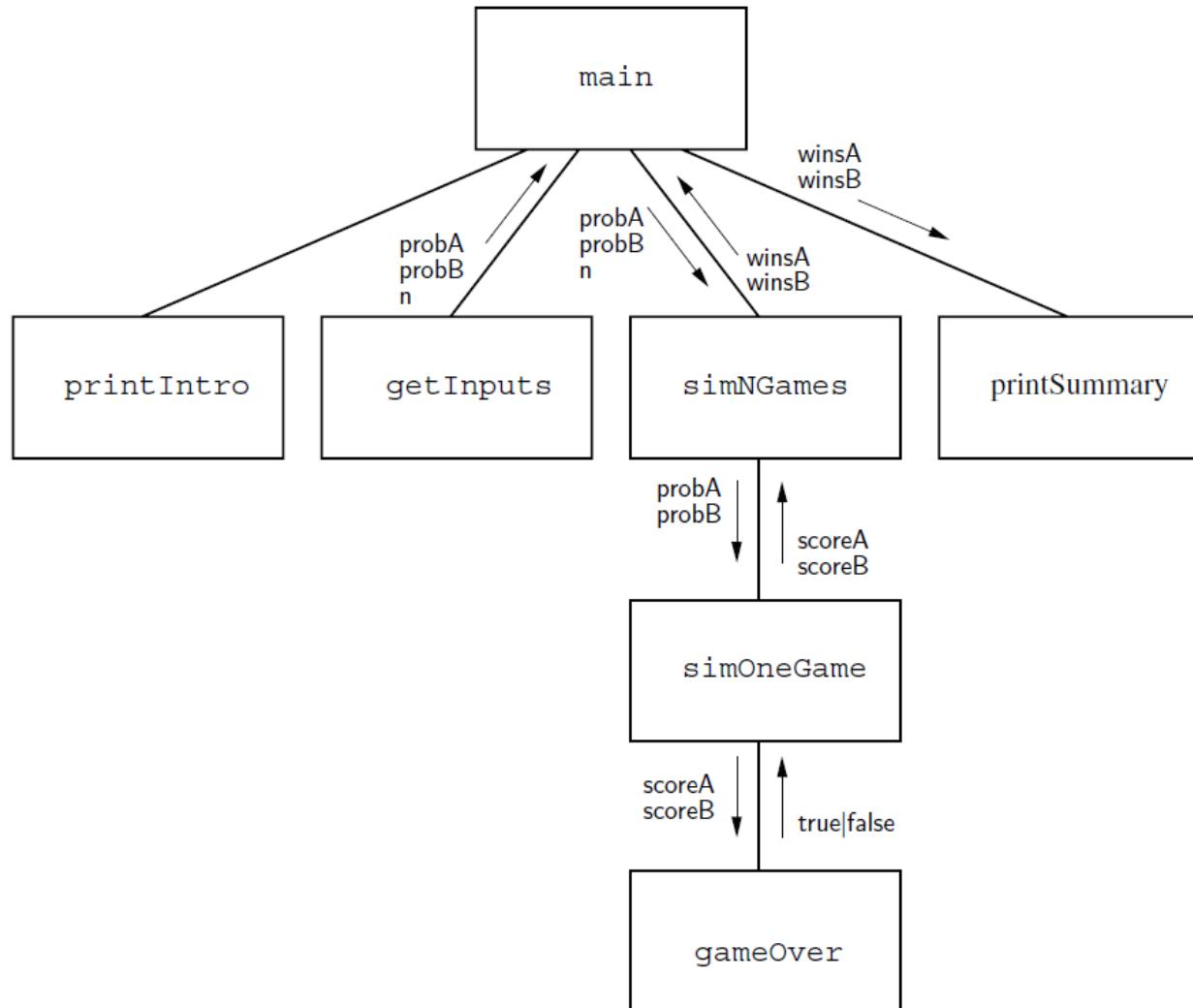
- Simulate one serve of whichever player is serving
- update the status of the game

Return scores

```
def simOneGame(probA, probB):  
    scoreA = 0  
    scoreB = 0  
    serving = "A"  
    while <condition>:
```

- What will the condition be?? Let's take the two scores and pass them to another function that returns True if the game is over, False if not.

Third-Level Design



Third-Level Design

- At this point, simOneGame looks like this:

```
def simOneGame(probA, probB):  
    # Simulates a single game of racquetball between A, B  
    # RETURNS A's final score, B's final score  
    serving = "A"  
    scoreA = 0  
    scoreB = 0  
    while not gameOver(scoreA, scoreB):
```

- Inside the loop, we need to do a single serve. We'll compare a random number to the probability of winning to determine if the server wins the point: `random() < prob`
 - The probability we use is determined by who is serving, contained in the variable `serving`.
-

Third-Level Design

- If A is serving, then we use A's probability, and based on the result of the serve, either update A's score or change the service to B.

```
if serving == "A":  
    if random() < probA:  
        scoreA += 1  
    else:  
        serving = "B"
```

Third-Level Design

- Likewise, if it's B's serve, we'll do the same thing with a mirror image of the code.

```
if serving == "A":  
    if rnd.random() < probA:      # A wins the serve  
        scoreA += 1  
    else:                      # A loses service  
        serving = "B"  
  
else:  
    if rnd.random() < probB:      # B wins the serve  
        scoreB += 1  
    else:                      # B loses service  
        serving = "A"
```

Third-Level Design

```
def simOneGame(probA, probB):
    # Simulates a single game of racquetball between players A, B
    # RETURNS A's final score, B's final score
    serving = "A"
    scoreA = 0
    scoreB = 0
    while not gameOver(scoreA, scoreB):
        if serving == "A":
            if rnd.random() < probA:
                scoreA += 1
            else:
                serving = "B"
        else:
            if rnd.random() < probB:
                scoreB += 1
            else:
                serving = "A"
    return scoreA, scoreB
```

Finishing Up

There's just one tricky function left, `gameOver`. Here's what we know:

```
def gameOver(a,b):  
    # a and b are scores for players in a racquetball game  
    # RETURNS true if game is over, false otherwise
```

- According to the rules, the game is over when either player reaches 15 points. We can check for this with the boolean expression: `a==15 or b==15`

Finishing Up

So, the complete code for gameOver looks like this:

```
def gameOver(a,b):  
    # a and b are scores for players in a racquetball game  
    # RETURNS true if game is over, false otherwise  
    return a == 15 or b == 15
```

printSummary is equally simple!

```
def printSummary(winsA, winsB):  
    # Prints a summary of wins for each player.  
    n = winsA + winsB  
    print("\nGames simulated:", n)  
    print("Wins for A: {} ({:.1%})".format(winsA,winsA/n))  
    print("Wins for B: {} ({:.1%})".format(winsB, winsB/n))
```

- Notice % formatting on the output

Summary of the Design Process

- We started at the highest level of our structure chart and worked our way down.
- At each level, we began with a general algorithm and refined it into precise code.
- This process is sometimes referred to as **step-wise refinement**.

Summary of the Design Process

1. Express the algorithm as a series of smaller problems.
2. Develop an interface for each of the small problems.
3. Detail the algorithm by expressing it in terms of its interfaces with the smaller problems.
4. Repeat the process for each smaller problem.

Bottom-Up Implementation

- Even though we've been careful with the design, there's no guarantee we haven't introduced some silly errors.
- Implementation is best done in small pieces.
 - *Start with the functions you know you need to put together*

Unit Testing

- A good way to systematically test the implementation of a modestly sized program is to start at the lowest levels of the structure, testing each component as it's completed.
- For example, we can import our program and execute various routines/functions to ensure they work properly.

Unit Testing

- When testing, need to have reproducible behaviour.
That is, the program behaves the same way each time it is executed.
 - *For programs involving pseudo-random numbers, this means using seed functions to fix the starting point*

```
import random as rnd
TESTING = True
def main():
    if TESTING :
        rnd.seed(7) # For reproducible behaviour during testing
    printIntro()
    probA, probB, n = getInputs()
    winsA, winsB = simNGames(n, probA, probB)
    printSummary(winsA, winsB)
```

Unit Testing

- We could start with the gameOver function.

```
>>> import rball  
>>> rball.gameOver(0, 0)  
False  
>>> rball.gameOver(5, 10)  
False  
>>> rball.gameOver(15, 3)  
True  
>>> rball.gameOver(3, 15)  
True
```

Unit Testing

- Notice that we've tested gameOver for all the important cases.
 - *We gave it 0, 0 as inputs to simulate the first time the function will be called.*
 - *The second test is in the middle of the game, and the function correctly reports that the game is not yet over.*
 - *The last two cases test to see what is reported when either player has won.*

Unit Testing

Now that we see that
gameOver is working,
we can go on to
simOneGame.

```
>>> simOneGame(0.5, 0.5)
(11, 15)
>>> simOneGame(0.5, 0.5)
(13, 15)
>>> simOneGame(0.3, 0.3)
(11, 15)
>>> simOneGame(0.3, 0.3)
(15, 4)
>>> simOneGame(0.4, 0.9)
(2, 15)
>>> simOneGame(0.4, 0.9)
(1, 15)
>>> simOneGame(0.9, 0.4)
(15, 0)
>>> simOneGame(0.9, 0.4)
(15, 0)
>>> simOneGame(0.4, 0.6)
(10, 15)
>>> simOneGame(0.4, 0.6)
(9, 15)
```

Unit Testing

- Testing each component in this manner is called **unit testing**.
- Testing each function independently makes it easier to spot errors, and should make testing the entire program go more smoothly.
- Then need end-to-end, or **integration**, testing.

Simulation Results

- Is it the nature of racquetball that small differences in ability lead to large differences in final score?
- Suppose Denny wins about 60% of his serves and his opponent is 5% better. How often should Denny win?
- Let's do a sample run where Denny's opponent serves first.

Simulation Results

This program simulates a game of racquetball between two players called "A" and "B". The abilities of each player is indicated by a probability (a number between 0 and 1) that the player wins the point when serving. Player A always has the first serve.

What is the prob. player A wins a serve? .65

What is the prob. player B wins a serve? .6

How many games to simulate? 5000

Games simulated: 5000

Wins for A: 3329 (66.6%)

Wins for B: 1671 (33.4%)

- With this small difference in ability , Denny will win only 1 in 3 games!

Other Design Techniques

- Top-down design is not the only way to create a program!

Prototyping and Spiral Development

- Another approach to program development is to start with a simple version of a program, and then gradually add features until it meets the full specification.
- This initial stripped-down version is called a **prototype**. (The method is sometimes called **rapid prototyping**.)

Prototyping and Spiral Development

- Prototyping often leads to a **spiral** development process.
- Rather than taking the entire problem and proceeding through specification, design, implementation, and testing, we first design, implement, and test a prototype.
 - *Basis of the Agile design methodologies*
- We take many mini-cycles through the development process as the prototype is incrementally expanded into the final program.
 - *At each step, consult with the client*

Prototyping and Spiral Development

- How could the racquetball simulation been done using spiral development?
 - *Write a prototype where you assume there's a 50-50 chance of winning any given point, playing 30 rallies.*
 - *Add on to the prototype in stages, including awarding of points, change of service, differing probabilities, etc.*

Prototyping and Spiral Development

```
from random import random

def simOneGame():
    scoreA = 0
    scoreB = 0
    serving = "A"
    for i in range(30):
        if serving == "A":
            if random() < 0.5:
                scoreA += 1
            else:
                serving = "B"
        else:
            if random() < 0.5:
                scoreB += 1
            else:
                serving = "A"
    print(scoreA, scoreB)
```

Ask yourself: is the function/program
doing sensible things?

```
>>> simOneGame()
0 0
0 1
0 1
...
2 7
2 8
2 8
3 8
3 8
3 8
3 8
3 8
3 8
3 9
3 9
4 9
5 9
```

Prototyping and Spiral Development

- The program could be enhanced in phases:
 - *Phase 1: Initial prototype. Play 30 rallies where the server always has a 50% chance of winning. Print out the scores after each server.*
 - *Phase 2: Add two parameters to represent different probabilities for the two players.*
 - *Phase 3: Play the game until one of the players reaches 15 points. At this point, we have a working simulation of a single game.*
 - *Phase 4: Expand to play multiple games. The output is the count of games won by each player.*
 - *Phase 5: Build the complete program. Add interactive inputs and a nicely formatted report of the results.*

Prototyping and Spiral Development

- Spiral development is useful when dealing with new or unfamiliar features or technology.
- If top-down design isn't working for you, try some spiral development!

The Art of Design

- Spiral development is not an alternative to top-down design as much as a complement to it – when designing the prototype you'll still be using top-down techniques.
- Good design is as much creative process as science, and as such, there are no hard and fast rules.
- The best advice?
 - *Three words*
 1. Practice
 2. Practice
 3. Practice





THE UNIVERSITY OF
WESTERN
AUSTRALIA

Lecture 13

Objects

Objectives of this Lecture

- To get familiar with Objects
- To understand the concept of objects and how they can be used to simplify programs
- Understand that in Python, everything is actually an object
- To get familiar with the various objects available in the graphics library
- To be able to create objects in programs
 - *call appropriate methods to perform graphical computations*

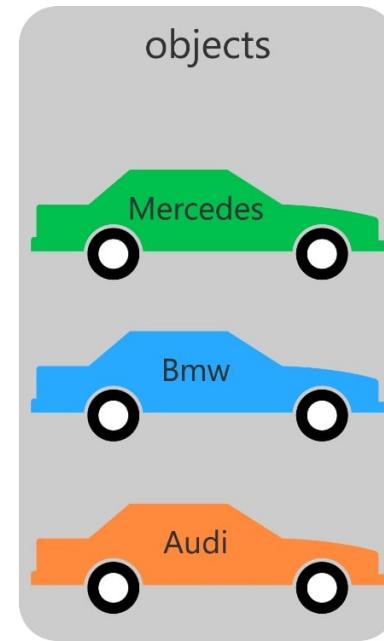
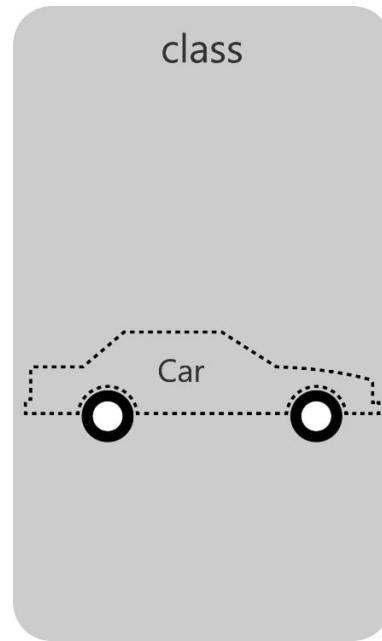
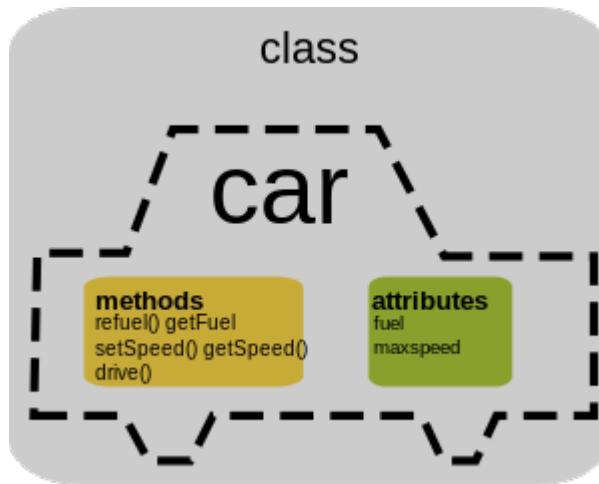
Overview

- So far, we saw that each data type can represent a certain set of values, and each had a set of associated operations.
- The traditional programming view is that data is passive – it is manipulated and combined using active operations.
- Modern computer programs are built using an **object-oriented** approach.

Objects and Object Oriented Programming

- Basic idea – view a complex system as the interaction of simpler **objects**.
- An object is a kind of active data type that combines data and operations.
 - *Objects know stuff (contain data) and they can do stuff (have operations).*
- Objects interact by sending each other messages (*requests do to stuff*).

OOP concept



Other Examples

Class

Definition of objects that share structure, properties and behaviours.



Building
class



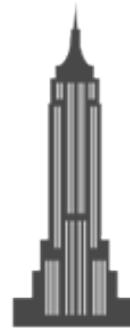
Dog
class



Computer
class

Instance

Concrete object, created from a certain class.



Empire State
instance of Building

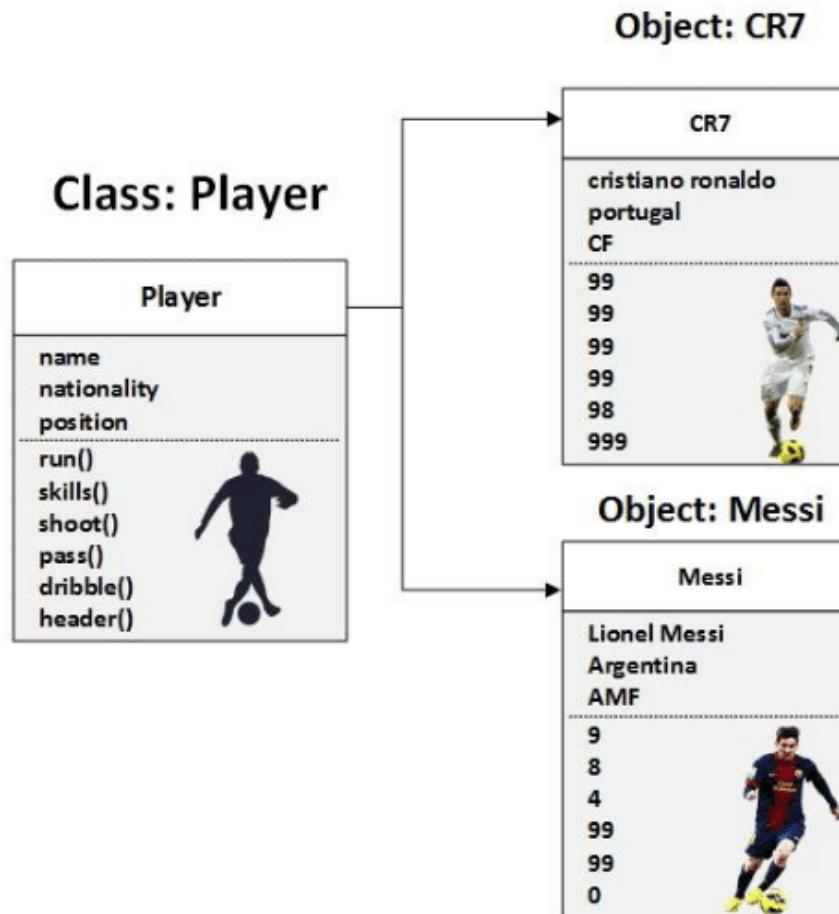


Lassie
instance of Dog



Your computer
instance of Computer

Example (2)



How to learn OOP using football

Objects: Explained with an Example

- Suppose we want to develop a data processing system for a university.
- We must keep records on students who attend the university.
- Each student, each unit, etc., will be represented as different sorts of objects.

University Student Object

- What information would be in a student object?
 - *Name*
 - *Home address*
 - *Residential address (if away from home)*
 - *Units*

What would the student object do?

- The student object should respond to requests.
- We may want to send out a campus-wide mailing, so we need a campus address for each student.
- We could send the `getHomeAddress()` to each student object. When the student object receives the message, it responds with the home address.

Course Object

- Each course might also be represented by an object:
- The Course-object:
 - *Instructor*
 - *Students enrolled*
 - *Pre-requisite courses*
 - *When and where the class meets*

Objects within Objects

- An object can have one or more objects inside it
- For example, the course-object will have student-objects inside
- Similarly, the course-object may have an instructor-object.

Sample operations of the Course-object

- addStudent ()
 - *Student-object added to course-object*
- delStudent ()
- changeRoom ()
- The point is that different operations are appropriate for objects (like different data-types)

Objects for Graphics Programming

- Most applications you're familiar with have **Graphical User Interfaces** (GUI)
- GUI provides windows, icons, buttons and menus (these are also known as objects).
- There's a simple graphics library written specifically to go with your text book.
- Operations using this library will be used to illustrate object-oriented programming in Python

Aside: Importing Library Functions

- Many Python programmers believe it is tedious to prepend library names in front of library functions, objects, etc,
 - `math.sqrt()`
- Python allows you to import all functions from a module
 - `from math import *`
All the functions from this library will be imported and can be used without further qualification.
 - `sqrt(5)` # rather than `math.sqrt(5)`

Importing Library Functions

- We can also import one function from a library

```
>>> from math import sqrt  
>>> sqrt(5)
```

- Problem is that after the import, further down the program, when you see the name of a function you have no idea where it came from.
 - *Can make debugging harder later*
- Better to leave original module name, or create shorthand:

```
>>> import math as  
>>> win = m.sqrt(5)
```

Simple Graphics Programming

- Python provides graphics capabilities through Tkinter.
- Your text book comes with `graphics.py` library
 - <http://mcsp.wartburg.edu/zelle/python/graphics.py>
 - *Copy on LMS*
- Where to put the library
 - *In the same folder as your other Python programs for this unit*

Using the graphics.py Library

- We need to import the library first

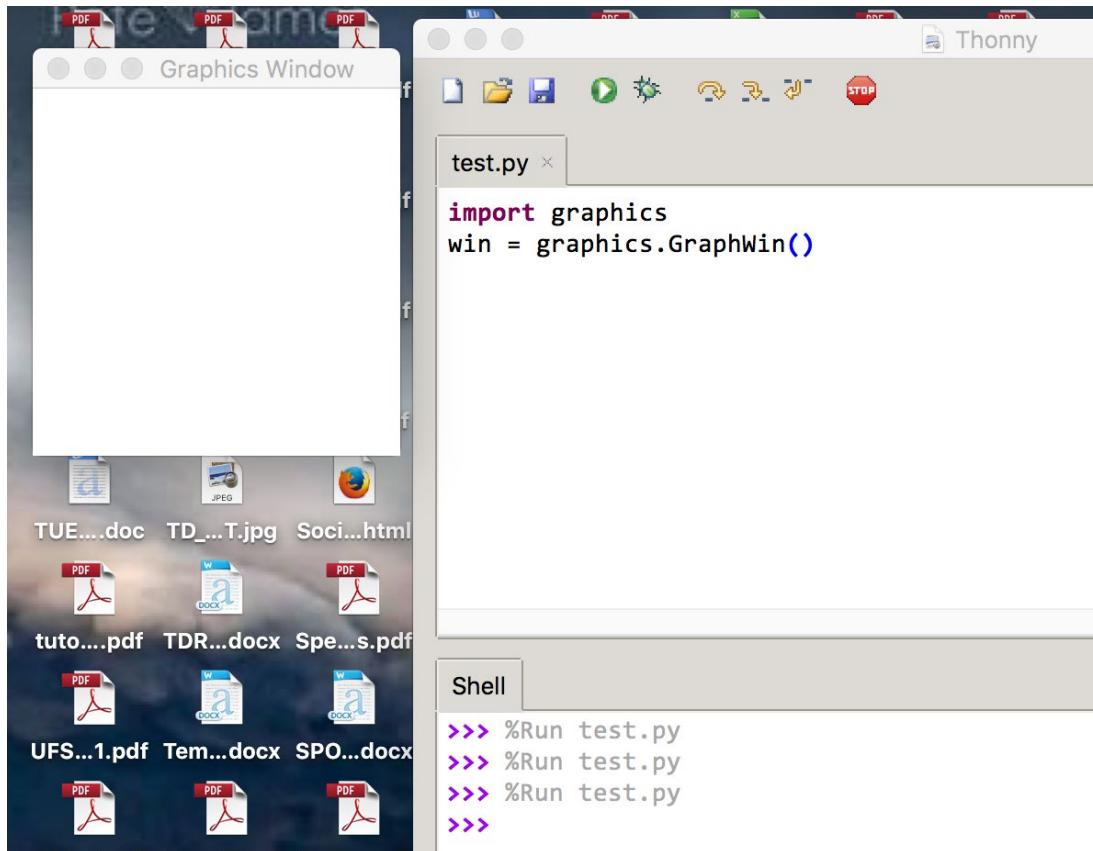
```
>>> import graphics
```

- A graphics window is a place on the screen where the graphics will appear.

```
>>> win = graphics.GraphWin()
```

- This command creates a new window object titled “Graphics Window”

Using the graphics.py Library



Graphics and Objects

- GraphWin () creates an object which is assigned to the variable *win*.
- We can manipulate the window object through this variable.
 - *Like having `x = 6` and then performing integer operations, e.g. `x *= 7`*
- For example, windows can be closed/destroyed by issuing the command

```
>>> win.close()
```

Graphics Window

- A graphics window is a collection of points called **pixels** (picture elements).
- The default GraphWin is 200 pixels tall by 200 pixels wide (40,000 pixels total).
- One way to get pictures into the window is one pixel at a time, which would be tedious.
- The graphics library has a number of predefined routines to draw geometric shapes.

A Point in Graphics

- The simplest object is the Point.
- Point locations are represented with a coordinate system (x, y) , where x is the horizontal location of the point and y is the vertical location.
- The origin $(0,0)$ in a graphics window is the upper left corner.
- X values increase from left to right, y values **from top to bottom**.
- Lower right corner is $(199, 199)$

Simple Graphics Commands

The screenshot shows a Python development environment with two windows. On the left is a 'Graphics Window' showing two small black dots at coordinates (50, 60) and (140, 100). On the right is a code editor window titled 'points.py' containing the following code:

```
import graphics as gx
p = gx.Point(50, 60)
print(p.getX())
print(p.getY())
win = gx.GraphWin()
p.draw(win)
p2 = gx.Point(140, 100)
p2.draw(win)
```

Below the code editor is a 'Shell' window showing the output of running the script:

```
>>> %Run test.py
>>> %Run points.py
50.0
60.0
>>>
```

A red arrow points from the text '(50,60)' in the bottom-left corner to the first point in the graphics window. Another red arrow points from the line 'p2.draw(win)' in the code editor to the second point in the graphics window.

Objects only become visible when the object is drawn in the window

Drawing Geometric Shapes

```
import graphics as gx

### Open a graphics window
win = gx.GraphWin('Shapes')

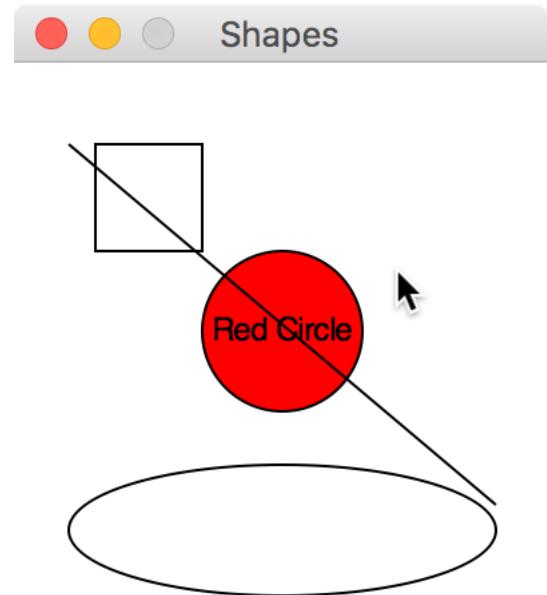
### Draw a red circle centered at point (100, 100) with
radius 30
center = gx.Point(100, 100)
circ = gx.Circle(center, 30)
circ.setFill('red')
circ.draw(win)

### Put a textual label in the center of the circle
label = gx.Text(center, "Red Circle")
label.draw(win)

### Draw a square using a Rectangle object
rect = gx.Rectangle(gx.Point(30, 30), gx.Point(70, 70))
rect.draw(win)

### Draw a line segment using a Line object
line = gx.Line(gx.Point(20, 30), gx.Point(180, 165))
line.draw(win)

### Draw an oval using the Oval object
oval = gx.Oval(gx.Point(20, 150), gx.Point(180, 199))
oval.draw(win)
```



Using Graphics Objects

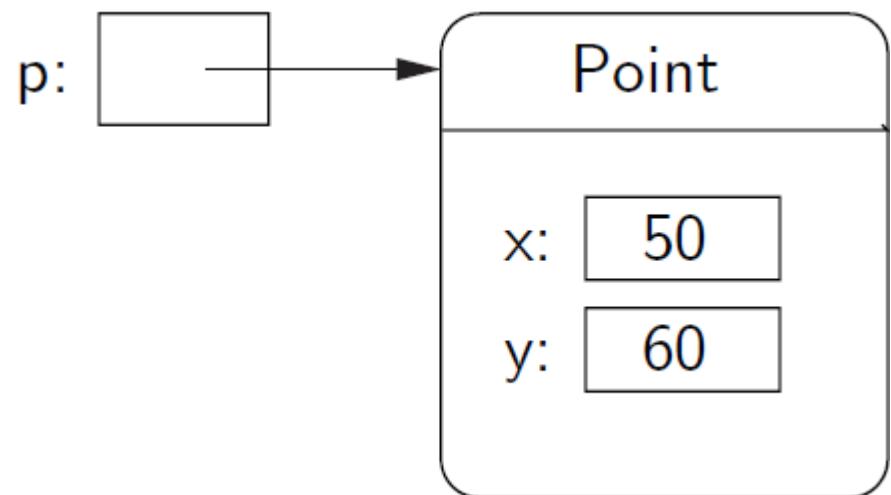
- Computation is preformed by asking an object to carry out one of its operations; “message”.
- In the previous example we manipulated GraphWin, Point, Circle, Oval, Line, Text and Rectangle. These are examples of *classes*.
- Each object is an instance of some **class** and the **class** describes the properties of the instance.
 - int, float, str, None *are classes*
- If we say Snoopy is a dog, we mean Snoopy is a specific individual of the **class** of dogs. Snoopy is an **instance** of the dog **class**.

Creating a New Instance

- To create a new instance of a class, we use a special operation called a *constructor*.
`<class-name>(<param1>, <param2>, ...)`
- A *<class-name>* is the name of the class we want to create a new instance of, e.g. Circle or Point.
- *The parameters are required to initialize the object. For example, Point requires two numeric values; GraphWin can, optionally, take a name for the window.*
 - `Point(50, 60)`

Example of Creating a New Instance

- `p = Point(50, 60)`
- The constructor for the `Point` class requires two parameters, the `x` and `y` coordinates for the point.
- These values are stored as *instance variables* inside of the object.



Class – Instance - Object

Class: Think of it as a “template” or a “blueprint” used to create objects.

Instance: A unique copy of a Class representing an Object.

Object: An Object is an Instance of a Class. It knows stuff and can do stuff.

Summary

- We learned some basics of Object Oriented programming
- We learned what are objects and how to use them in our programs
- We learned the difference between classes, instances and objects
- We learned how to write simple graphics programs
- We haven't learned how to define our own classes yet. This will be covered in a few weeks time.



THE UNIVERSITY OF
WESTERN
AUSTRALIA

Lecture 18

Search algorithms

Objectives

- To know what searching is and understand the algorithms for linear and binary search.
- To understand the basic techniques for analyzing the efficiency of algorithms.

Searching

- Searching is the process of looking for a particular value in a collection.
- For example, a program that maintains a membership list for a club might need to look up information for a particular member – this involves a search process.
 - *Range from simple search in a list to search across huge databases, e.g. Google search*

Simple Searching

- Here is the specification of a simple searching function:

```
def search(x, nums):  
    # nums is a list of numbers and x is a number  
    # Returns the position in the list where x occurs  
    # or None if x is not in the list.
```

- Here are some sample interactions

```
>>> search(4, [3, 1, 4, 2, 5])  
2  
>>> search(7, [3, 1, 4, 2, 5])  
None
```

Simple Searching Python

- The Boolean list method `in` tests for list membership

`x in nums`

- Use `index()` to find position of item in a list

```
>>> nums = [3, 1, 4, 2, 5]
```

```
>>> nums.index(4)
```

```
2
```

- Only problem is that the `index` method raises an exception if the sought item is not present

Example: Find Julia and Mandy

- | | | |
|---------------|---------------|---------------|
| 1. Sophia | 35. Nevaeh | 69. Autumn |
| 2. Isabella | 36. Kaylee | 70. Jocelyn |
| 3. Emma | 37. Alyssa | 71. Faith |
| 4. Olivia | 38. Anna | 72. Lucy |
| 5. Ava | 39. Sarah | 73. Stella |
| 6. Emily | 40. Allison | 74. Jasmine |
| 7. Abigail | 41. Savannah | 75. Morgan |
| 8. Madison | 42. Ashley | 76. Alexandra |
| 9. Mia | 43. Audrey | 77. Trinity |
| 10. Chloe | 44. Taylor | 78. Molly |
| 11. Elizabeth | 45. Brianna | 79. Madelyn |
| 12. Ella | 46. Aaliyah | 80. Scarlett |
| 13. Addison | 47. Riley | 81. Andrea |
| 14. Natalie | 48. Camila | 82. Genesis |
| 15. Lily | 49. Khloe | 83. Eva |
| 16. Grace | 50. Claire | 84. Ariana |
| 17. Samantha | 51. Sophie | 85. Madeline |
| 18. Avery | 52. Arianna | 86. Brooke |
| 19. Sofia | 53. Peyton | 87. Caroline |
| 20. Aubrey | 54. Harper | 88. Bailey |
| 21. Brooklyn | 55. Alexa | 89. Melanie |
| 22. Lillian | 56. Makayla | 90. Kennedy |
| 23. Victoria | 57. Julia | 91. Destiny |
| 24. Evelyn | 58. Kylie | 92. Maria |
| 25. Hannah | 59. Kayla | 93. Naomi |
| 26. Alexis | 60. Bella | 94. London |
| 27. Charlotte | 61. Katherine | 95. Payton |
| 28. Zoey | 62. Lauren | 96. Lydia |
| 29. Leah | 63. Gianna | 97. Ellie |
| 30. Amelia | 64. Maya | 98. Mariah |
| 31. Zoe | 65. Sydney | 99. Aubree |
| 32. Hailey | 66. Serenity | 100. Kaitlyn |
| 33. Layla | 67. Kimberly | |
| 34. Gabriella | 68. Mackenzie | |

Linear Search

- Say you are given a page full of randomly ordered numbers and are asked whether 13 is in the list.
- You may start at the front of the list, comparing each number to 13
- If you see it, you can say that it is in the list. If you have scanned the whole list and not seen it, you will tell me it isn't there.
- This is called **linear search**.

Linear Search

```
def search(x, nums):  
    for i in range(len(nums)):  
        if nums[i] == x: # item found  
            return i #return index value  
    return None #loop finished, item not in list
```

- This algorithm wasn't hard to develop, and works well for modest-sized lists
- The Python `in` and `index` operations both implement linear searching algorithms.
- If the collection of data is very large, it makes sense to organize the data somehow so that each data value doesn't need to be examined.
 - *Avoid non-solutions*

Find Omar and Otto

2012	2013	2014	2015	2016	2017
Alberto	Andrea	Arthur	Ana	Alex	Arlene
Beryl	Barry	Bertha	Bill	Bonnie	Bret
Chris	Chantal	Cristobal	Claudette	Collin	Cindy
Debby	Dorian	Dolly	Danny	Danielle	Don
Ernesto	Erin	Edouard	Erika	Earl	Emily
Florence	Fernand	Fay	Fred	Fiona	Franklin
Gordon	Gabrielle	Gonzalo	Grace	Gaston	Gert
Helene	Humberto	Hanna	Henri	Hermine	Harvey
Isaac	Ingrid	Isayas	Ida	Ian	Irma
Joyce	Jerry	Josephine	Joaquin	Julia	Jose
Kirk	Karen	Kyle	Kate	Karl	Katia
Leslie	Lorenzo	Laura	Larry	Lisa	Lee
Michael	Melissa	Marco	Mindy	Matthew	Maria
Nadine	Nestor	Nana	Nicholas	Nicole	Nate
Oscar	Olga	Omar	Odette	Otto	Ophelia
Patty	Pablo	Paulette	Peter	Paula	Philippe
Rafael	Rebekah	Rene	Rose	Richard	Rina
Sandy	Sebastien	Sally	Sam	Shary	Sean
Tony	Tanya	Teddy	Teresa	Tobias	Tammy
Valerie	Van	Vicky	Victor	Virginie	Vince
William	Wendy	Wilfred	Wanda	Walter	Whitney

Strategy 1: Linear Search

- If the data is sorted in ascending order (lowest to highest), we can skip checking some of the data.
- As soon as a value is encountered that is greater than the target value, the linear search can be stopped without looking at the rest of the data.
- On average, this will save us about half the work.

Strategy 2: Binary Search

- If the data is sorted, there is an even better searching strategy
 - one you probably already know!
- Have you ever played the number guessing game, where I pick a number between 1 and 100 and you try to guess it? Each time you guess, I'll tell you whether your guess is correct, too high, or too low. What strategy do you use? How many maximum number of guesses are required?
- Each time we guess the middle of the remaining numbers to try to narrow down the range.
- This strategy is called ***binary search***, because at each step we are dividing the remaining group of numbers into two parts.

Strategy 2: Binary Search

- We can use the same approach in our binary search algorithm! We can use two variables to keep track of the endpoints of the range in the sorted list.
- Since the target could be anywhere in the list, initially `low` is set to the first location in the list, and `high` is set to the last.
- The heart of the algorithm is a loop that looks at the middle element of the range, comparing it to the value x .
- If x is smaller than the middle item, `high` is moved so that the search is confined to the lower half.
- If x is larger than the middle item, `low` is moved to narrow the search to the upper half.

Strategy 2: Binary Search

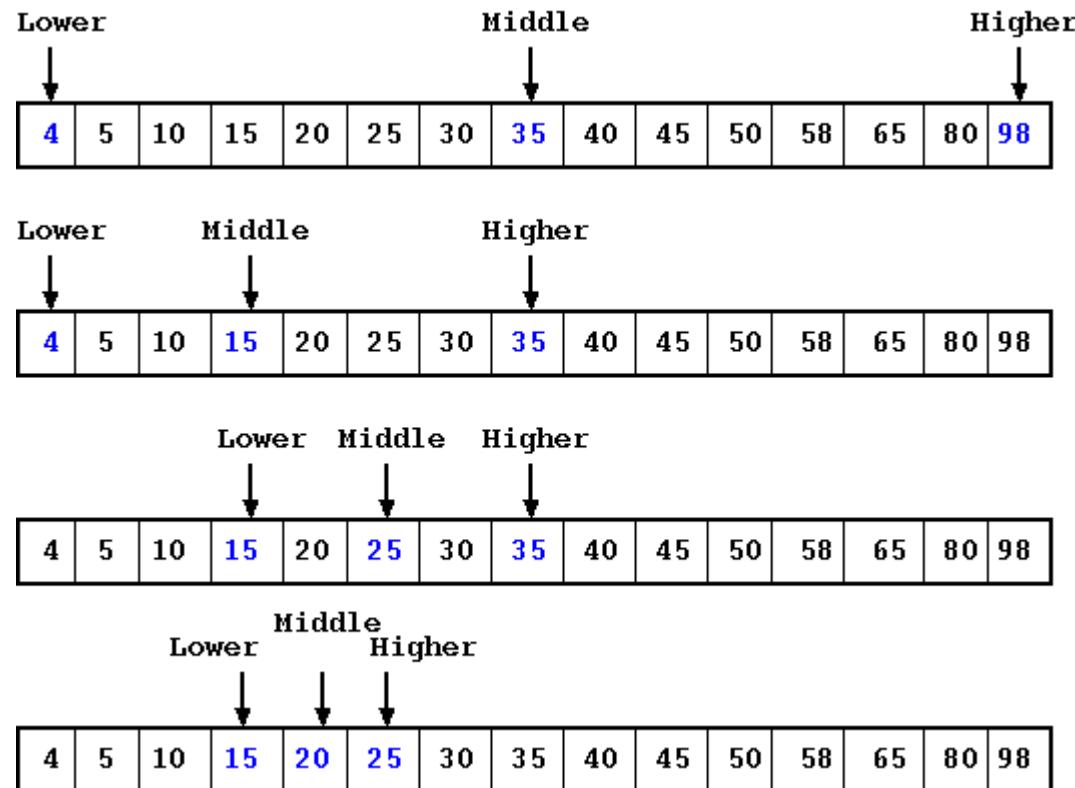
- We can use the same approach in our binary search algorithm! We can use two variables to keep track of the endpoints of the range in the sorted list.
 - Since the target could be anywhere in the list, initially `low` is set to the first location in the list, and `high` is set to the last.
 - The heart of the algorithm is a loop that looks at the middle element of the range, comparing it to the value x .
 - If x is smaller than the middle item, `high` is moved so that the search is confined to the lower half.
 - If x is larger than the middle item, `low` is moved to narrow the search to the upper half.
-

Strategy 2: Binary Search

- The loop terminates when either
 - x is found
 - There are no more places to look ($low > high$)

Example: Search for 20 in this list of numbers.

What happens if you search for 7?



Strategy 2: Binary Search

```
def search(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:           # There is still a range to search
        mid = (low + high)//2   # Position of middle item
        item = nums[mid]
        if x == item:           # Found it! Return the index
            return mid
        if x < item:            # x is in lower half of range
            high = mid - 1     # move top marker down
        else:                   # x is in upper half of range
            low = mid + 1       # move bottom marker up
    return -1                  # No range left to search,
                                # x is not there
```

Linear vs Binary Search

- Which search algorithm is better, linear or binary?
 - *The linear search is easier to understand and implement*
 - *The binary search is more efficient since it doesn't need to look at each element in the list*
- Intuitively, we might expect the linear search to work better for small lists, and binary search for longer lists. But how can we be sure?
 - *Experiment*

Linear vs Binary Search

- Test program searches 100 times for a integer in a list of 1,000,000 integers
 - *Using linear and binary search*
 - *When randomly chosen integer present, and when random integer not present in list*

Search when item is in list

Linear search

User: 10.35 Sys: 0.01

Binary search

User: 0.00 Sys: 0.00

Search for item not in list

Linear search

User: 15.52 Sys: 0.02

Binary search

User: 0.00 Sys: 0.00

Comparing Algorithms

- Empirical results are dependent on the type of computer they were conducted on, the amount of memory in the computer, the speed of the computer, etc.?
- We could abstractly reason about the algorithms to determine how efficient they are. We can assume that the algorithm with the fewest number of “steps” is more efficient.
- How do we count the number of “steps”?
- Computer scientists attack these problems by analyzing the number of steps (very approximately) that an algorithm will take relative to the size or difficulty of the specific problem instance being solved.

Comparing Algorithms

- 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?

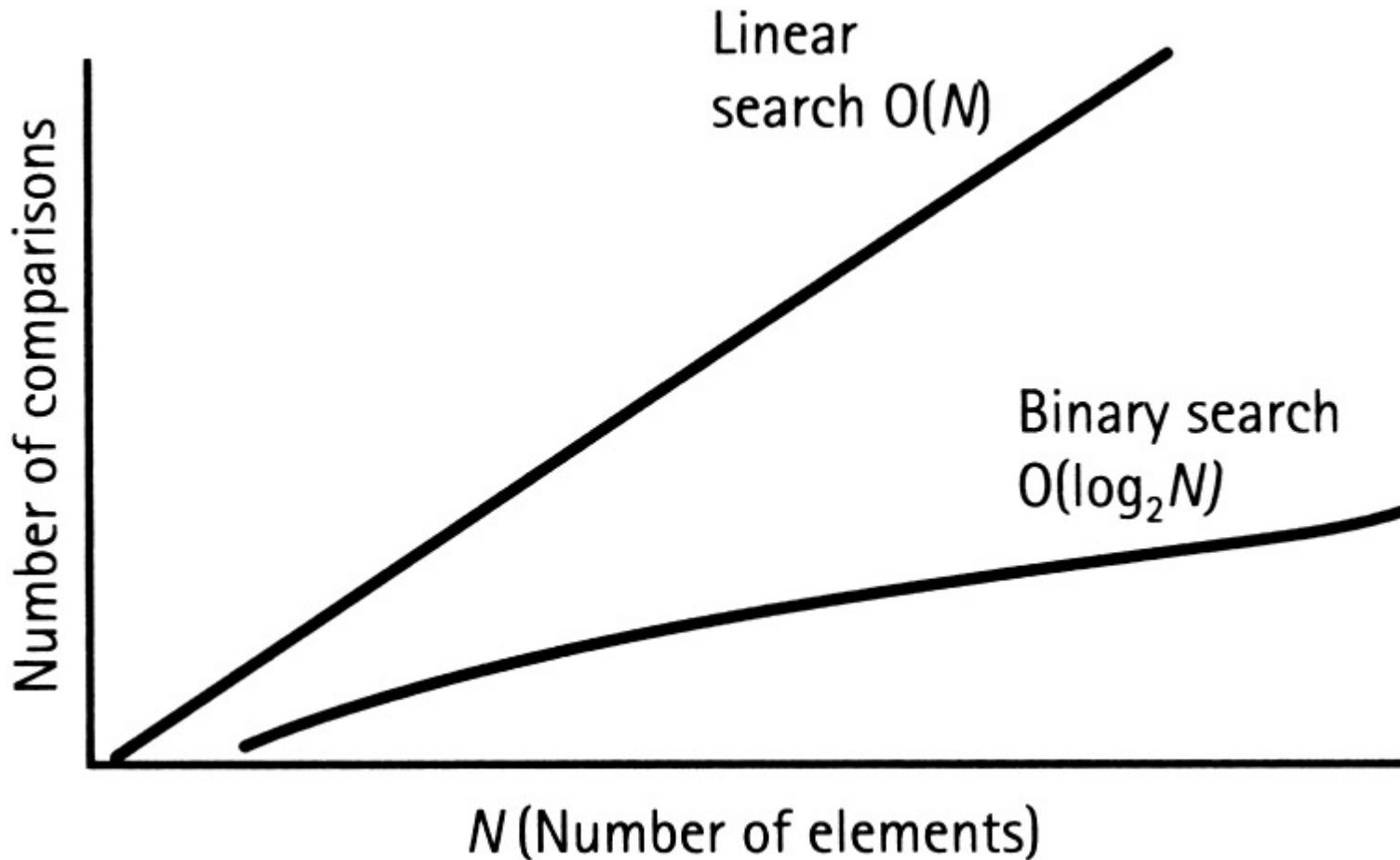
Comparing Algorithms

- Let's consider linear search.
 - *For a list of 10 items, the most work we might have to do is to look at each item in turn – looping at most 10 times.*
 - *For a list twice as large, we would loop at most 20 times.*
 - *For a list three times as large, we would loop at most 30 times!*
- The amount of time required is linearly related to the size of the list, n . This is what computer scientists call a *linear time* algorithm.
 - *Notation used is $O(N)$ – order notation*

Comparing Algorithms

- Now, let's consider binary search
 - Suppose the list has 16 items. Each time through the loop, half the items are removed from the search. After one loop, 8 items remain to be searched.
 - After two loops, 4 items remain.
 - After three loops, 2 items remain
 - After four loops, 1 item remains.
 - If a binary search loops i times, it can find a single value in a list of size 2^i .
 - Put another way, if the list has size N , $i = \log_2(N)$ loops will be required. $O(\log(N))$ time
 - Approx 20 loops for 1,000,000 item list
-

Comparing Algorithms



<https://www.techtud.com/computer-science-and-information-technology/algorithms/searching/binary-search>

Comparing Algorithms

- Earlier, I mentioned that Python uses linear search in its built-in searching methods. Why doesn't it use binary search?
 - *Binary search requires the data to be sorted*
 - *If the data is unsorted, it must be sorted first!*
 - You will learn higher level programming courses that sorting takes $O(N \times \log(N))$ so not worth the trouble

Binary Search overview

- 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 sub-problems 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.

Summary

- Understood and coded Linear search
- Understood and coded Binary search
- Comparing efficiency of algorithms



THE UNIVERSITY OF
WESTERN
AUSTRALIA

Lecture 19

Recursion

Objectives

- To understand recursion
- To understand when to use recursion
- Recursion vs iteration

Recursive Problem-Solving

Algorithm: Factorial – find the factorial for x

```
if n == 0:  
    return the factorial of zero [0! = 1]  
  
else  
    find factorial of (n-1)      [(n-1)!]  
    return n * (n-1)!
```

- This version has no loop, and seems to refer to itself!
What's going on??

Recursive Definitions

- A description of something that refers to itself is called a *recursive* definition.
- In the last example, the factorial algorithm uses its own description – a “call” to factorial “recurs” inside of the definition – hence the label “recursive definition.”

Recursive Definitions

- In mathematics, recursion is frequently used. The most common example is the factorial:
- For example, $5! = 5(4)(3)(2)(1)$, or
 $5! = 5(4!)$

$$n! = n(n-1)(n-2)\dots(1)$$

Recursive Definitions

- In other words, $n! = n(n - 1)!$
- Or

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{otherwise} \end{cases}$$

- This definition says that $0!$ is 1, while the factorial of any other number is that number times the factorial of one less than that number.

Recursive Definitions

- Our definition is recursive, but definitely not circular. Consider $4!$
 - $4! = 4(4-1)! = 4(3!)$
 - *What is $3!$? We apply the definition again*
 $4! = 4(3!) = 4[3(3-1)!] = 4(3)(2!)$
 - *And so on...*
 $4! = 4(3!) = 4(3)(2!) = 4(3)(2)(1!) = 4(3)(2)(1)(0!) = 4(3)(2)(1)(1) = 24$
- Factorial is not circular because we eventually get to $0!$, whose definition does not rely on the definition of factorial and is just 1. This is called a *base case* for the recursion.
- When the base case is encountered, we get a closed expression that can be directly computed.

Recursive Definitions

- All good recursive definitions have these two key characteristics:
 1. *There are one or more base cases for which no recursion is applied.*
 2. *All chains of recursion eventually end up at one of the base cases.*

Put differently, each iteration must drive the computation toward a base case.
- 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 Functions

- Factorial can be calculated using a loop accumulator.

```
def fact_loop(n):  
    ans = 1  
  
    for i in range(n, 1, -1):  
        ans *= i  
  
    return ans
```

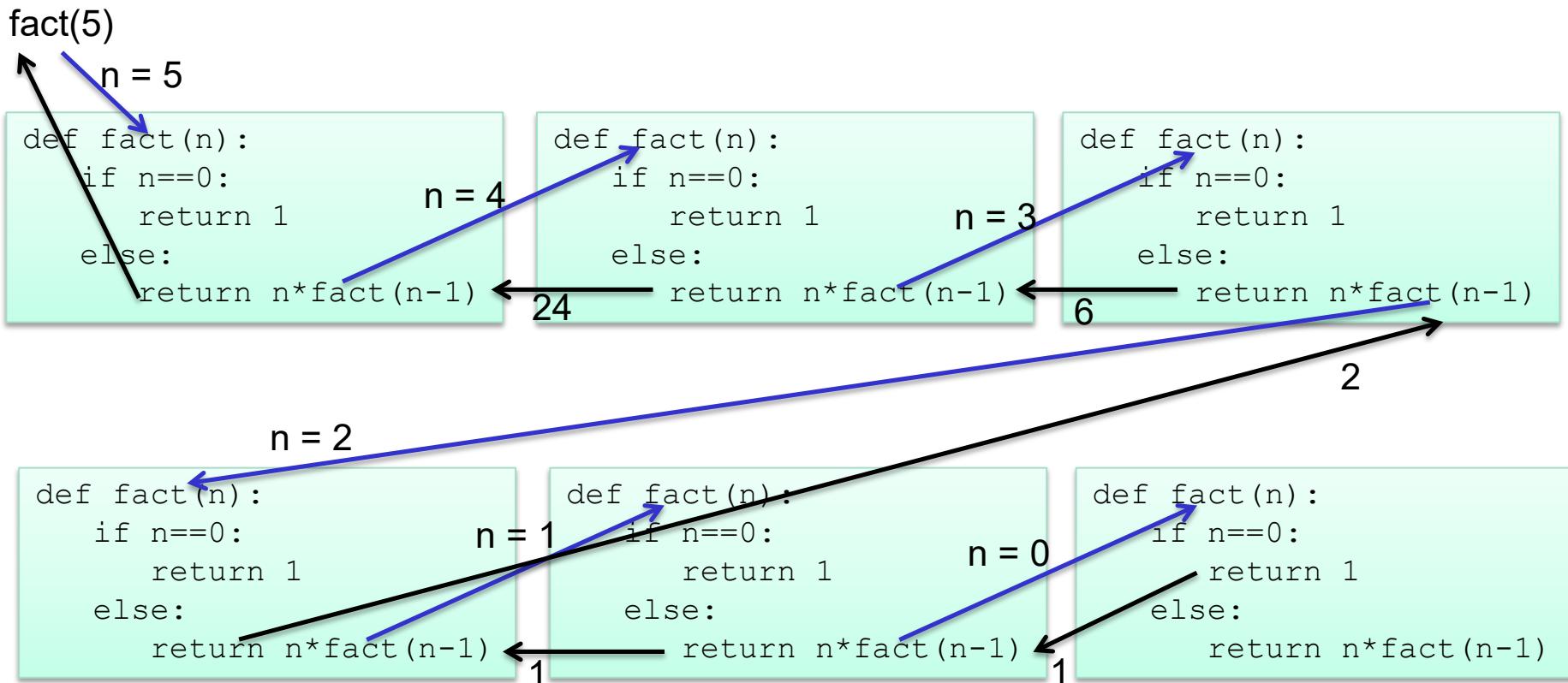
- If factorial is written as a separate recursive function:

```
def fact(n):  
    if n == 0:  
        return 1  
  
    else:  
        return n * fact(n-1)
```

Recursive Functions

- We've written a function that calls *itself*, i.e. a *recursive function*.
- The function first checks to see if we're at the base case ($n==0$) . If so, return 1. Otherwise, return the result of multiplying n by the factorial of $n-1$, $\text{fact}(n-1)$.
- Remember that each call to a function starts that function anew, with its own copies of local variables and parameters.

Recursive Functions



Example: Binary Search

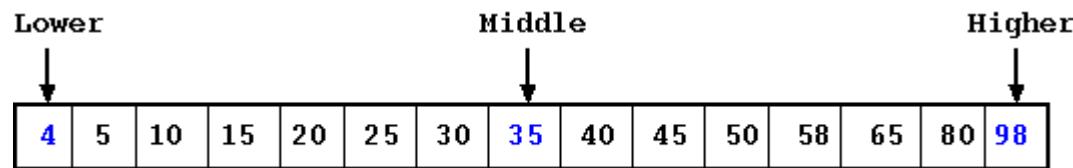
- In the last lecture, we learned how to perform binary search using a loop.
- If you haven't noticed already, we can perform binary search recursively.
- In binary search, we look at the middle value first, then we either search the lower half or upper half of the array.
- There are two base cases (to stop recursion/searching):
 - *when the target value is found*
 - *when we have run out of places to look.*

Example: Binary Search

- The recursive calls will cut the search in half each time by specifying the range of locations that are not searched and may contain the target value.
- Each invocation of the search routine will search the list between the given *low* and *high* parameters.

Example: Binary Search

```
def recBinSearch(x, nums, low, high):
    if low > high:                      # No place left to look, return -1
        return -1
    mid = (low + high) // 2
    item = nums[mid]
    if item == x:
        return mid
    if x < item:                         # Look in lower half
        return recBinSearch(x, nums, low, mid-1)
                                            # Look in upper half
    return recBinSearch(x, nums, mid+1, high)
```



We can then call the binary search with a generic search wrapping function

```
def search(x, nums):
    return recBinSearch(x, nums, 0, len(nums)-1)
```

Recursion vs. 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!
 - *But some algorithms harder to set up with iteration*
- Some programming languages use recursion exclusively.
 - *Haskell, ML, Lisp (functional programming); Prolog (logic programming)*
- Some problems that are simple to solve with recursion are quite difficult to solve with loops.

Recursion vs. Iteration

- In the factorial and binary search problems, the looping and recursive solutions use roughly the same algorithms, and their efficiency is nearly the same.
- Lets take another example: Fast Exponentiation

Example: Fast Exponentiation

- One way to compute a^n for an integer n is to multiply a by itself n times.
- This can be done with a simple accumulator loop:

```
def loopPower(a, n):  
    ans = 1  
    for i in range(n):  
        ans *= a  
    return ans
```

Example: Fast Exponentiation

- We can also solve this problem using recursion and divide & conquer approach.
- Using the laws of exponents, we know that $2^8 = 2^{4 \times 2^4}$. If we know 2^4 , we can calculate 2^8 using one multiplication.
- What's 2^4 ? $2^4 = 2^2 \times 2^2$, and $2^2 = 2 \times 2$.
- $2 \times 2 = 4$, $2^2 \times 2^2 = 16$, $2^4 \times 2^4 = 256 = 2^8$
- We've calculated 2^8 using only three multiplications!

Example: Fast Exponentiation

- We can take advantage of the fact that $a^n = a^{n//2}(a^{n//2})$
- This algorithm only works when n is even. How can we extend it to work when n is odd?
- $2^9 = 2^4 \times 2^4 \times 2^1$

$$a^n = \begin{cases} a^{n//2}(a^{n//2}) & \text{if } n \text{ is even} \\ a^{n//2}(a^{n//2})(a) & \text{if } n \text{ is odd} \end{cases}$$

Example: Fast Exponentiation

- This method relies on integer division (if n is 9, then $n//2 = 4$).
- To express this algorithm recursively, we need a suitable base case.
- If we keep using smaller and smaller values for n , n will eventually be equal to 0 ($1//2 = 0$), and $a^0 = 1$ for any value except $a = 0$.

Example: Fast Exponentiation

```
# raises a to the int power n
def recPower(a, n):
    if n == 0:
        return 1
    factor = recPower(a, n//2)
    if n%2 == 0:      # n is even
        return factor * factor
    # n is odd
    return factor * factor * a
```

- Here, a temporary variable called `factor` is introduced so that we don't need to calculate $a^{n/2}$ more than once, simply for efficiency.

Recursion vs. Iteration

- In the exponentiation problem:
 - *The iterative version takes linear time to complete*
 - *The recursive version executes in log time.*
 - *The difference between them is like the difference between a linear and binary search.*
- So... will recursive solutions always be as efficient or more efficient than their iterative counterpart?
- It depends

Recursion vs. Iteration

- The Fibonacci sequence is the sequence of numbers 1,1,2,3,5,8,...
 - *The sequence starts with two 1's*
 - *Successive numbers are calculated by finding the sum of the previous two numbers.*

Recursion vs. Iteration

```
def loopfib(n):  
    # returns the nth Fibonacci number  
    curr = 1  
    prev = 1  
    for i in range(n-2):  
        curr, prev = curr+prev, curr  
    return curr
```

- Note the use of simultaneous assignment to calculate the new values of `curr` and `prev`.
- The loop executes only $n-2$ times since the first two values have already been provided as a starting point.

Recursion vs. Iteration

- The Fibonacci sequence also has a recursive definition:

$$fib(n) = \begin{cases} 1 & \text{if } n < 3 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

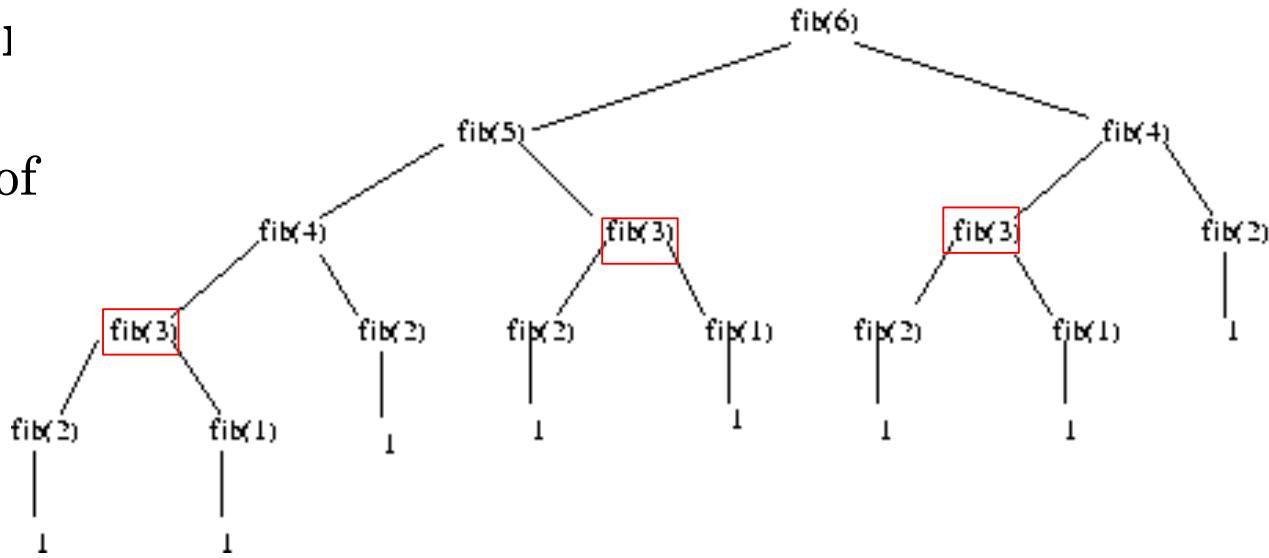
- This recursive definition can be directly turned into a recursive function!

```
def fib(n):  
    if n < 3:  
        return 1  
    return fib(n-1)+fib(n-2)
```

Recursion vs. Iteration

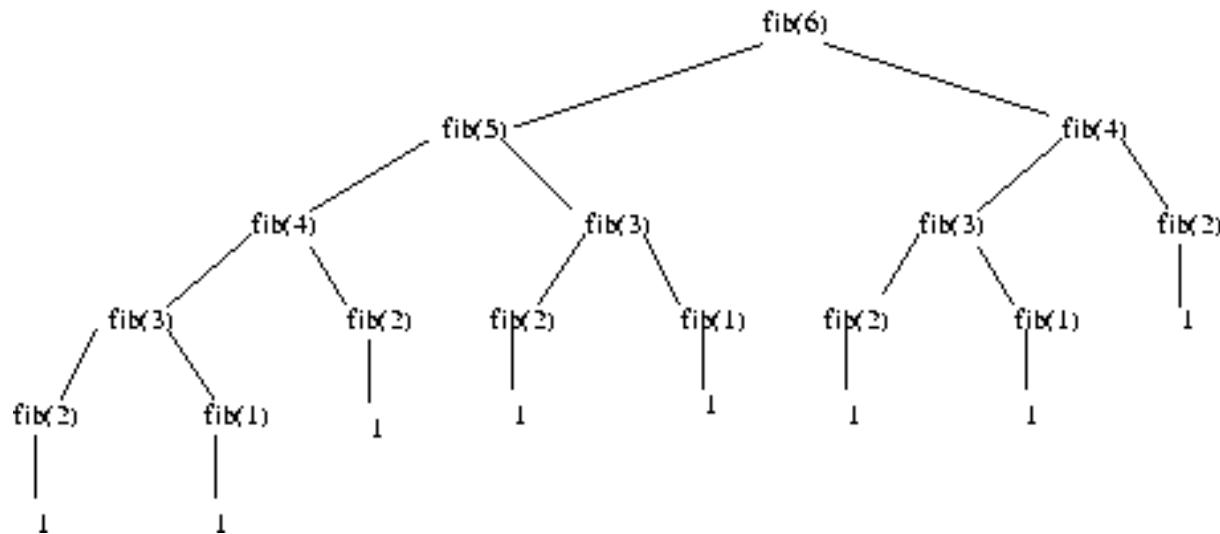
- This function obeys the rules that we've set out.
 - *The recursion is always based on smaller values.*
 - *There is a non-recursive base case.*
- So, this function will work great, won't it? – *Sort of...*
- The recursive solution is extremely inefficient, since it performs]

Recomputing of
 $\text{fib}(3)$ shown



Recursion vs. Iteration

- To calculate $\text{fib}(6)$, $\text{fib}(4)$ is calculated twice, $\text{fib}(3)$ is calculated three times, $\text{fib}(2)$ is calculated four times... For large numbers, this adds up!



Recursion vs. Iteration

- 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 and (generally) simplicity of programming
- Avoid the recursive solution if it is terribly inefficient, unless you can't come up with an iterative solution (which sometimes happens!)

Summary

- We learned the concept of recursion.
- We analyzed its performance and compared it to iterations (loops)
- We learned when to use recursion and when to use loops