**Computer Science 61a:**

**Week 1**

## 1.1   Getting Started

- CS has affected all parts of human life

-  All computing begins with representing information, specifying logic to process it, and designing abstractions that manage the complexity of that logic.

**1.1.1   Programming in Python**

- We will use Python

- Guido Van Rossum first conceived and implemented Python

- Python is very popular and easy to use

- Python code learn..

- The best way to learn is to start experimenting

**1.1.2   Installing Python 3**

- Download Python 3 (http://getpython3.com/diveintopython3/installing-python.html)

**1.1.3   Interactive Sessions**

- In an interactive session (IS), you type code after >>>, then python executes.

- To start IS run Python by typing python3 at a terminal prompt.

- Ex

- Sessions are recorded.

Ctrl.P (Previous) – Ctrl.N (Next) – Ctrl.D (Exit) – Also use up and down arrows

**1.1.4   First Example**

- Ex

- [**>>> from** **urllib.request** **import** urlopen] is an import statement to locate a URL by making available a function called urlopen

**Statements & Expressions**:

* Computer programs can:

1. Compute some value
2. Carry out some action

* Statements describe actions
* Expressions describe computations
* [**>>>** shakespeare = urlopen('http://composingprograms.com/shakespeare.txt')] associated the name Shakespeare with the value of the expression that follows =.

**Functions:**

* Functions encapsulate logic that manipulates data.
* urlopen is a function. A web address is a piece of data, and the text of Shakespeare's plays is another.
* Ex

**Objects:**

* An object seamlessly bundles together data and the logic that manipulates that data.
* A set is an object
* Ex

**Interpreters:**

- A program that implements the procedure of evaluating compound expressions

- Functions are objects, objects are functions, and interpreters are instances of both; they are all connected

**1.1.5   Errors**

* Errors occurs a lot in programming
* Computers are powerful and stupid
* Learning to interpret errors and diagnose the cause of unexpected errors is called *debugging*
* Guidelines:

1. **Test incrementally**: test each small bit of your code individually.
2. **Isolate errors**: trace the error to the smallest fragment of code you can before trying to correct it.
3. **Check your assumptions**: Know your assumptions, and then focus your debugging effort on verifying that your assumptions actually hold.
4. **Consult others:** Ask!

## 1.2   Elements of Programming

* Programs are written for people to read
* Ways that the language provides to combine simple ideas to build complex tasks:

1. **Primitive expressions and statements**, the simplest building blocks that the language provides
2. **Means of combination**, by which compound elements are built from simpler ones,
3. **Means of abstraction**, by which compound elements can be named and manipulated as units.

* In programming, we deal with two kinds of elements: functions and data.
* Data: things we want to manipulate.
* Functions: rules for manipulating the data.

**1.2.1   Expressions**

* A number is a primitive expression, which can be combined with mathematical operators.
* *infix* notation: where the *operator* (e.g., +, -, \*, or /) appears in between the *operands* (numbers).

### 1.2.2   Call Expressions

* A call expression applies a function to some arguments
* The *operator* is an expression that precedes parentheses, which enclose a comma-delimited list of *operand* expressions.
* Ex
* The operator specifies a *function*. When this call expression is evaluated, we say that the function x is *called* with the operands as *arguments* and, this all, *is returned as a value.*
* The order of the arguments in a call expression matters.
* The function pow raises its first argument to the power of its second argument.
* Function notation has three principal advantages over the mathematical convention of infix notation.

1. Functions may take an arbitrary number of arguments with no ambiguity because the function’s name always precedes the argument.
2. Function notation extends in a straightforward way to *nested* expressions, which is all specified by parentheses.
3. Functions are simpler to type.

**1.2.3   Importing Library Functions**

* Python has a lot of defined elements (expressions) in it’s library that can be used by importing them first.

Python organizes the expressions, in the library, in modules e.g. math module.

* sqrt function takes the square root
* The operator module provides access to functions corresponding to infix operators e.g. add, sub, mul.
* Ex
* An import statement designates a module name (e.g., operator or math), and then lists the named attributes of that module to import (e.g., sqrt). Once a function is imported, it can be called multiple times.
* List of the functions defined by each module (<https://docs.python.org/3/library/index.html>)

**1.2.4   Names and the Environment**

* If a value has been given a name, we say that the name *binds* to the value.
* In Python, we can establish new bindings using the assignment statement, which contains a name to the left of = and a value to the right
* Ex
* Names are also bound via import statements
* The = symbol is called the assignment operator in Python
* Assignment is the simplest way of abstraction.
* Complex programs are constructed by building, step by step, computational objects of increasing complexity.
* An *environment* is a memory in which python keeps track of names values and bindings.
* Names can also be bound to functions.
* Python prints an identifying description instead of a text, when asked to describe a function because its hard.
* We can use assignment statements to give new names to existing functions, and each successive assignment rebinds the name to new values.
* In Python, names are often called *variable names* or *variables* because they can be bound to different values in the course of executing a program.
* When a name is bound to a new value through assignment, it is no longer bound to any previous value.
* When executing an assignment statement, Python evaluates the expression to the right of = before changing the binding to the name on the left. Therefore, one can refer to a name in right-side expression, even if it is the name to be bound by the assignment statement. Also, swapping the values bound to two names can be performed in a single statement.
* Ex’s
* We can also assign multiple values to multiple names in a single statement, where names on the left of =and expressions on the right of = are separated by commas.

**1.2.5   Evaluating Nested Expressions**

* To evaluate a call expression, Python will do the following:

1. Evaluate the operator and operand subexpressions
2. Apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpressions.

* The evaluation procedure is *recursive* in nature
* Ex, is an expression tree.
* Trees grow from the top down
* Nodes are the objects at each point on a tree ( in this case, they are expressions paired with their values.)
* Rules for evaluating primitive expressions:

1) A numeral evaluates to the number it names,

2) A name evaluates to the value associated with that name in the current environment.

- Environments provide the context in which evaluation takes place

- Each type of expression or statement has its own evaluation or execution procedure.

- Statements are not evaluated but *executed; they do not produce a value but instead make some change.*

***1.2.6   The Non-Pure Print Function***

* There are two types of functions:

1. **Pure functions.** Functions have some input (their arguments) and return some output (the result of applying them). E.g. abs(x).

 Pure functions have the property that applying them has no effects beyond returning a value. Moreover, a pure function must always return the same value when called twice with the same arguments.

1. **Non-pure functions.** In addition to returning a value, applying a non-pure function can generate side effects e.g. generate additional output beyond the return value using the *print* function.

* The value that print returns is always None, a special Python value that represents nothing.
* The interactive Python interpreter does not automatically print the value None.
* In the case of print, the function itself is printing output as a side effect of being called.
* Be careful with print! The fact that it returns None means that it *should not* be the expression in an assignment statement.
* Pure funcitons are beneficial because they are more reliable and simpler to test, they are also essential for writing *concurrent* programs, in which multiple call expressions may be evaluated simultaneously.

**1.3   Defining New Functions**

* Function definition is a powerful abstraction technique by which a name can be bound to compound operation, which can be referred to then as a unit.
* Ex
* A formal parameter is what should be inputted as a value for the operator can apply on.

**How to define a function**

1. A  def statement that indicates a <name>
2. A comma-separated list of named <formal parameters
3. A return statement, called the function body, that specifies the <return expression> of the function, which is an expression to be evaluated whenever the function is applied.

* Ex

def <name>(<formal parameters>):

return <return expression>

* The second line *must* be indented (4 spaces)
* The return expression is not evaluated right away; it is stored as part of the newly defined function and evaluated only when the function is eventually applied.
* User-defined functions are used in exactly the same way as built-in functions.
* Both def statements and assignment statements bind names to values, and any existing bindings are lost.

### 1.3.1   Environments

- An environment in which an expression is evaluated consists of a sequence of *frames*, depicted as boxes. Each frame contains *bindings*, each of which associates a name with its corresponding value. There is a single *global* frame. Assignment and import statements add entries to the first frame of the current environment.

- Ex (imp)

- Each function is a line that starts with func, followed by the function name and formal parameters. Built-in functions such as mul do not have formal parameter names, and so ... is always used instead.

* The name of a function is repeated twice, once in the frame and again as part of the function itself.
* The name appearing in the function is called the *intrinsic name*. The name in a frame is a *bound name*.
* There is a difference between the two: different names may refer to the same function, but that function itself has only one intrinsic name
* The error message TypeError: 'int' object is not callable is reporting that the name max is an integer and not a function. (so it cant be an operator)

**Function Signatures**

- Functions differ in the number of arguments they’re allowed to take.

- The description of the formal parameters of a function is its signature.

- The function max can take an arbitrary number of arguments. It is rendered as max(...). Regardless of the number of arguments taken, all built-in functions will be rendered as <name>(...), because these primitive functions were never explicitly defined.

### 1.3.2   Calling User-Defined Functions

- The interpreter evaluates u-d functions the same way it does other functions.

- Applying a user-defined function introduces a second *local* frame, which is only accessible to that function.

- To apply a user-defined function to some arguments:

1) Bind the arguments to the names of the function's formal parameters in a new *local* frame.

2) Execute the body of the function in the environment that starts with this frame.

- The body of a function is not executed until the function is called (not when it is defined).

- The "Return value" in the square() frame is not a name binding; instead it indicates the value returned by the function call that created the frame.

- The order of frames in an environment affects the value returned by looking up a name in an expression.

**Name Evaluation.**

* A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

### 1.3.3   Example: Calling a User-Defined Function

- Ex (Refer to the section)

- A new local frame is introduced every time a function is called, even if the same function is called twice.

-  Local frames keep the names separate.

### 1.3.4   Local Names

-  The meaning of a function should be independent of the parameter names chosen by its author

-  The simplest consequence is that the parameter names of a function must remain local to the body of the function.

- The *scope* of a local name is limited to the body of the user-defined function that defines it.

- This scoping behavior isn't a new fact about our model; it is a consequence of the way environments work.

### 1.3.5   Choosing Names

* Guidelines for your code:

1. Function names are lowercase, with words separated by underscores. Descriptive names are encouraged.
2. Function names typically evoke operations applied to arguments by the interpreter (e.g., print,add, square) or the name of the quantity that results (e.g., max, abs, sum).
3. Parameter names are lowercase, with words separated by underscores. Single-word names are preferred.
4. Parameter names should evoke the role of the parameter in the function, not just the kind of argument that is allowed.
5. Single letter parameter names are acceptable when their role is obvious, but avoid "l" (lowercase ell), "O" (capital oh), or "I" (capital i) to avoid confusion with numerals.

- There are many inconsistencies still.

**1.3.6   Functions as Abstractions**

- A function definition suppresses details and provides abstraction; thus, any two functions that provides the same result should be indistinguishable.

**Aspects of a functional abstraction**

* Three core attributes to master functional abstraction:

1) The *domain* of a function is the set of arguments it can take

2) The *range* of a function is the set of values it can return

3) The *intent* of a function is the relationship it computes between inputs and output (+ side effects)

- These attributes do not specify how the intent is carried out; that detail is abstracted away.

**1.3.7   Operators**

* They are evaluated like call expressions.
* They can be nested and are evaluated according to pedmas.
* / returns (explicit) floats
* // returns (round down) integers
* These two operators are shorthand for the truediv and floordiv functions.

## 1.4   Designing Functions

- What makes a good function (the qualities of good functions all reinforce the idea that functions are abstractions.)

(Please refer to the section)

1. Functions should have only one job.
2. *Don't repeat yourself* is a central tenet of software engineering.
3. Functions should be defined generally

**1.4.1   Documentation**

* A function definition will often include documentation describing the function, called a *docstring*, which must be indented along with the function body. Docstrings are conventionally triple quoted. The first line describes the job of the function in one line. The following lines can describe arguments and clarify the behavior of the function
* Ex
* When you call help with the name of a function as an argument, you see its docstring (type q to quit Python help).
* **>>>** help(pressure)

**Comments**

* Comments in Python can be attached to the end of a line following the # symbol.

**1.4.2   Default Argument Values**

* A function may have a default argument that is ignored if argument value is provided.
* Ex
* As a guideline, most data values used in a function's body should be expressed as default values to named arguments

**Week 2**

## 1.5   Control

* Control statements control the flow of a program's execution based on the results of logical comparisons.
* Statements have no value; they determine what the interpreter should do next.

**1.5.1   Statements**

- Types of Statements:

1) assignment (=) 2) def 3) return

- Each statement describes some change to the interpreter state, and executing a statement applies that change.

- Expressions can also be executed as statements, by discarding their value.

- If you don’t return an expression their will be no value

- Ex

- You don’t have to return non-pure functions e.g. print

**1.5.2   Compound Statements**

-  A simple statement is a single line that doesn't end in a colon.

- A compound statement is so called because it is composed of other statements (simple and compound).

- A Clause is a header and an indented suit of statements

- Ex

- Simple statements examples:

1) Expressions 2) Return statements 3) Assignment statements

- A def statement is a compound statement.

- Multi-line programs work in this way:

To execute a sequence of statements, execute the first statement. If that statement does not redirect control, then proceed to execute the rest of the sequence of statements, if any remain.

* The execution rule is applied recursively
* Later statements may never be reached because of redirected control

**1.5.3   Defining Functions II: Local Assignment**

* The process of function application terminates whenever the first return statement is executed, and value is returned
* The effect of an assignment statement is to bind a name to a value in the *first* frame of the current environment, thus, they cannot affect the global frame

**1.5.4   Conditional Statements**

- **Conditional statements:** consists of a series of headers and suites: a required if clause, an optional sequence of elif clauses, and finally an optional else clause.

- Ex

- The computational process of executing a conditional clause:

1) Evaluate the header's expression.

2) If it is a true value, execute the suite. Then, skip over all subsequent clauses in the conditional statement.

- **Boolean contexts:**

- The expressions inside the header statements of conditional blocks are said to be in *boolean context.*

* Their truth matters to flow but their values are not returned
* Python includes several false values, including 0, None, and the *boolean* value False. All other numbers are true values
* Every built-in kind of data in Python has both true and false values

**Boolean values**. Python has two boolean values, True and False.

- Boolean values represent truth values in logical expressions.

- The built-in comparison operations, >, <, >=, <=, ==, !=, return these values.

- ge is greater than or equal to is >=

- eq is equal to is ==

**Boolean operators**. Three basic logical operators are also built into Python:

**>>> True** **and** **False**

False

**>>> True** **or** **False**

True

**>>> not** **False**

True

* Ex (imp)
* Functions that perform comparisons and return boolean values typically begin with is, not followed by an underscore (e.g., isfinite, isdigit, isinstance, etc.).

### 1.5.5   Iteration

* Control statements are used to express repetition.

- A while clause contains a header expression followed by a suite:

while <expression>:

<suite>

-To execute a while clause:

1) Evaluate the header's expression.

2) If it is a true value, execute the suite, then return to step 1

- In step 2, the entire suite of the while clause is executed before the header expression is evaluated again.

- A while statement that does not terminate is called an infinite loop. Press <Control>-C to force Python to stop looping.

### 1.5.6   Testing

* Tests typically take the form of another function that contains one or more sample calls to the function being tested.
* **Assertions.** Programmer’s uses assert statements to verify expectations, such as the output of a function being tested.
* An assert statement has an expression in a boolean context, followed by a quoted line of text that will be displayed if the expression evaluates to a false value.
* Ex

**Doctests.** Python provides a convenient method for placing simple tests directly in the docstring of a function

* Ex

**>>> from** **doctest** **import** testmod

**>>>** testmod()

TestResults(failed=0, attempted=2)

* To verify the doctest interactions for only a single function, we use a doctest function called run\_docstring\_examples
* It’s first argument is the function to test; second is the result of the expression globals() ; third is True
* Ex
* If the value is incorrect their will be a test failure.
* When writing Python in files, all doctests in a file can be run by starting Python with the doctest command line option:

python3 -m doctest <python\_source\_file>

## 1.6   Higher-Order Functions

* Functions provide the ability to build abstractions by assigning names to common patterns and then to work in terms of the names directly.
* To express certain general patterns as named concepts, we will need to construct functions that can accept other functions as arguments or return functions as values.
* Functions that manipulate functions are called higher-order functions.

### 1.6.1   Functions as Arguments

* A pattern for computing sums:
* def <name>(n):

total, k = 0, 1

while k <= n:

total, k = total + <term>(k), k + 1 return total

* A generic higher-order function way of summing:

**>>> def** summation(n, term):

total, k = 0, 1

**while** k <= n:

total, k = total + term(k), k+1

**return** total

Then

**>>> def** Func(x):

**return** x

Then

**>>> def** sum\_Func(n):

**return** summation(n, Func)

### 1.6.2   Functions as General Methods

* The improve function which refines guesses to certain numbers:

**>>> def** improve(update, close, guess=1):

**while** **not** close(guess):

guess = update(guess)

**return** guess

-

- Compound functions all take one argument