Introduction to Python

1 Introduction

The purposes of this course are:

- Learn computational models of thinking.
- Master the art of computational problem solving.
- Make computers do what you want them to do.

1.1 What does a computer do?

Fundamentally, a computer does only two things:

1. It performs calculations.
2. It remember results.

One might be wondering how fast a computer performs calculations, so nowadays a computer can perform approximately a billion calculations per second. On the other hand, nowadays computers have a hard drive of nearly 1 TB which means that a computer can store approximately 1.5 million books of standard size.

1.2 What kind of operations does a computer perform?

Every computer comes with a set of built-in operations. These are typically primitive arithmetic operations such as addition, multiplicaton, division, and simple logic operations such as comparing true and false values in order make decisions.

Because this built-in tools are not enough, in this course one will learn how to define new calculations, new operations, and give them to the computer, so it can abstract them, encapsulate them, and treat them as they are primitives.

1.3 Are simple calculations and storage enough?

It is very important to design good algorithms since simple taks could take a lot of time, and results could required a lot of space to be storaged. For example, searching on the world wide web using simple operations could take 5.2 days, or deciding a chess move without an optimal algorithm could take 30 minutes.

Considering the same chess problem, experts suggest that there are approximately 10^{123} different possible games. However, there are only 10^{80} atoms in the observable universe. Hence, it is simply not possible to store all different possible games in chess.

1.4 Are there limits?

Despite its speed and size, a computer has limitations. For instance, some problems are still too complex such as accurate weather prediction and cracking encryption schemes, or some ploblems are fundamentally impossible to compute as the famous Halting problem which basically asks to predict whether a piece of code will always halt with an answer for any input.

2 Types of Knowledge

It is possible to divide knowledge into two types:

1. *Declarative knowledge:* it is statements of fact, i.e., statements of truth.
2. *Imperative knowledge:* it is a recipe, i.e., how-to knowlege or how-to information. It gives a sequence of steps to find a solution.

Both kind of knowledge are important, but only the second one allows getting the computer to do something.

2.1 A numerical example

Declarative knowledge: the square root of a number x is y such that $y * y = x$. The previous sentence is a statement of fact. Though it is not telling how to find a square root, it is telling how to check if a number is the square root of another one.

Imperative knowledge: Heron of Alexandria's algorithm to find the square root of a number x (e.g. 16):

1. Start with a guess, g.
2. If $g * g$ is close enought to x, then stop and say g is the answer.

3. Otherwise, make a new guess by averaging g and x/g.
4. Using the new guess, repeat process until close enough.

g	$g * g$	x/g	$(g + x/g)/2$
3	9	5.333	4.1667
4.1667	17.36	3.837	4.0035
4.0035	16.0277	3.997	4.000002

Table 1: Heron's algorithm to find the square root of 16.

2.2 What is a recipe?

A recipe has:

1. A sequence of simple steps.
2. A flow of control, a process that specifies when each step is executed.
3. A way to decide when to stop.

The previous three pieces constitute what is known as an algorithm.

3 Machines

One might be wondering how to capture a recipe or algorithm in a mechanical process. Historically, there have been two options. The former is to use what is called as a *fixed program computer*, i.e., a computer specifically designed to calculate a particular computation. For example, handheld calculators have been fixed to do addition, subtraction, multiplication, and division. This is a simple set of arithmetic operations, and that's all they can do.

The alternative to a fixed program computer is a *stored program computer*. it is possible to load an algorithm into this computer kind of computer, and inside of it, a set of parts are going to execute those instructions whenever one wants. Inside a stored program computer there is a special program known as the interpreter which is going to walk through each of those sequence of instructions doing the required computation. The advantage is that now one can load any algorithm without the need of having to build another computer to run the new program.

3.1 Basic Machine Architecture

Basically, a machine has memory which is a place where to store information. This information could be data, but it can also be a sequence of instructions

that constitute an algorithm. A machine will also have a way to load things into it and to print data out of it, i.e., input and output.

Inside the heart of the machine there are two elements. The former is called A.L.U. (Arithmetic Logic Unit) which takes information from memory, reads it in to perform a primitive operation (as addition or multiplication), and then it will store stuff back up into memory. The latter is called control unit which keeps track of the specific operation that has to be computed in the A.L.U. at each point in time.

Inside the control unit there is an important element called program counter. When loading a program into a machine, the program counter points to the location of the first instruction. When one asks the machine to execute the program, the program counter reads that first instruction, so it will cause an operation to take place, and it will also add one to the program counter which is going to take it to the next instruction in the sequence and repeat the process. At some point, a test might be reached, and the value of this test is going to change the program counter. Eventually, something is going to indicate to stop the program.

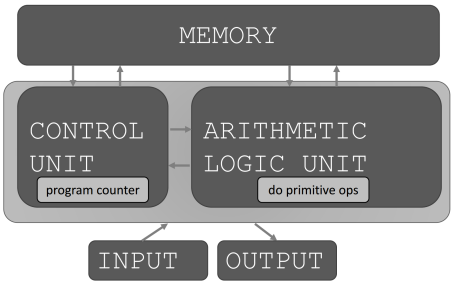

Figure 1: Basic Machine Architecture

3.2 Stored Program

A stored program is a sequence of instructions which is built out of simple arithmetic instructions, logic instructions and simple tests, and it also allows moving data around. This program has associated with it a special program called the interpreter which is going to execute each of those sequences of instructions in order according to the flow of control, and it will also terminate the program when indicated.

3.3 Basic Primitives

Most machines come with simple arithmetic and logic operations, but it is possible to go simpler than this. Alan Turing showed that one can compute anything just with six primitives, and this principle is known as Turing completeness. In fact, there is something called a Turing machine which is an infinite tape with

a set of squares on it. In each square, there is a symbol that could be either 0 or 1.

What Turing showed is that if you have six operations and those are: move left, move right, print, scan, erase, and do nothing, one can compute anything that is computable. Fortunately, programming languages come with a more convinient set of primitives, but the fundamental idea is that *with a simple set of primitives, one is able to compute anything.*

The power of computational thinking relies on being able to write descriptions, i.e., to abstract methods in order to create new primitives. There is a fundamental property that says that anything that is computable in one programming language is computable in another programming language, and this is also a proof of Turing completeness. What is different is that some taks could be easily done in some programming languages than in others.

4 Languages

4.1 Creating Recipes

A programming language provides a set of primitive operations, and the next step is to put them together. In order to to that, one uses expressions which are complex but legal combinations of primitives. Any legal expression in a programming language, any computation, has associated with it a value.

4.2 Aspects of Languages

While in natural languages such as English the primitive constructs are words, in a programming language the primitive constructs are *numbers*, sequences of characters a.k.a. *strings*, and *simple operations* such as addition, subtraction, and comparison.

Once again, when having those primitives, one has to put them together considering both syntax and semantics. *Syntax* tells whether or not an expression or combination of expresions is legal. Associated with every expression that is syntactically valid is the meaning a.k.a semantics. On one hand, *static semantics* tells which syntactically valid expression have meaning. On the other hand, *semantics* is the meaning associated with that syntactically correct string of symbols with no semantic errors. In order to clarify these three concepts *syntaxis, static semantics* and *semantics* see table 2.

Though in English it is possible to have multiple meaning for a sentence, as in the sentence *Tim saw Jay with a telescope*, in programming languages an expression has only one meaning. However, the meaning could be not the one that me programmer intended.

Sentence	Syntaxis	Static Sem.	Semantics
I have a dog.	✓	✓	✓
Tim saw Jay with a telescope.	✓	✓	✗
I are hungry	✓	✗	no eval
cat dog boy	✗	no eval	no eval

Table 2: Syntaxis vs. static semantics vs semantics

4.3 Where Things Go Wrong

Even when a programmer keeps in mind sintaxis, static semantics and semantics, things can go wrong. For example, he can encounter *syntactic errors* which are common but easy to find since most programming languages provide a way to catch them. In addition, a programmer might encounter *static semantic errors*, i.e., things that they are in the right order, but they don't make sense. Some programming languages check for them before running a program, but other languages, as Python, do it on the fly.

The bigger problem is when there are no semantic errors, but the programmer gets a different meaning than what he expected. There different possible consequences such as the program can crash or stop running, the program can run forever, or even worse, the program gives an answer that is different than the expected.

5 Types

As it has been previously mentioned, a program is a sequence of *definitions* and *commands*. Defitions means ways of either assigning names to values or more importantly creating procedures that will be treated as if they are primitives. Commands are simpler expressions that can be executed directly within Python using the shell which briefly speaking is a window into which one can type expressions that are passed to the Python interpreter and prints out the result.

5.1 Objects

One of the fundamental primitives in Python are *objects*. Objects represent data, and programs manipulate objects in order to get out data of them or do something with them. Every object has a type associated with it, and this type tells programs whether or not they can act on it.

Objects could be either *scalar* objects or *non-scalar* objects. Scalar objects cannot be subdivided, and non-scalar objects have internal structure into which a programmer can pull out parts.

5.2 Scalar Objects in Python

There are very few scalar objects in Python:

- `int` - represent integers as 9
- `float` - represent real numbers as 3.1415
- `bool` - represent boolean values as `True` or `False`
- `NoneType` - special one that has just one value `None`

It is possible to use `type()` to see the type of an object.

```
In [1]: type(9)
Out[1]: int

In [2]: type(3.1415)
Out[2]: float

In [3]: type(True)
Out[3]: bool

In [4]: type(False)
Out[4]: bool

In [5]: type(None)
Out[5]: NoneType
```

5.3 Type Conversion

It is possible to convert an object of one type to another. This is called casting.

```
In [1]: float(9)
Out[1]: 9.0

In [2]: type(float(9))
Out[2]: float

In [3]: int(3.1415)
Out[3]: 3

In [4]: type(int(3.1415))
Out[4]: int
```

Notice that in Python when casting from `float` to `int` the decimal part is truncated.

5.3.1 Printing to the console

It is possible to use the console or shell to evaluate expressions, but sometimes the programmer might want to print an expression, so he can use `print()`.

```
In [1]: 3 + 2
Out[1]: 5

In [2]: print(3 + 2)
5
```

Notice that when printing something there is no output since no value is returned. `print` returns `None`.

5.3.2 Expressions

In order to form expressions, which have a value which has a type, objects and operator must be combined. The following is the syntax for a simple expression.

```
<object> <operator> <object>
```

5.3.3 Operators on `int`s and `float`s

Considering that `i` and `j` are either an `int` or a `flaot`, these are the available operators:

- `i + j` $\longrightarrow$ the sum of `i` and `j`
- `i - j` $\longrightarrow$ the difference of `i` and `j`
- `i * j` $\longrightarrow$ the product of `i` and `j`
- `i / j` $\longrightarrow$ the division of `i` and `j`
- `i // j` $\longrightarrow$ the integer division of `i` and `j`
- `i % j` $\longrightarrow$ the remainder when`i` is divided by `j`
- `i ** j` $\longrightarrow$ the `j`-th power of `i`

```
In [1]: 3 - 2
Out[1]: 1

In [2]: 3 - 2.0
Out[2]: 1.0

In [3]: 3 / 2
Out[3]: 1.5
```

```
In [4]: 3 // 2
Out[4]: 1

In [5]: 5 % 3
Out[5]: 2
```

When both `i` and `j` have the same type, the result of the sum, difference, multiplication or power is of this type, but when mixing an `int` and a `float` the result is a `float`. The result of a division is always a `float`, while the result of the integer division is an `int` which is the quotient of that division.

5.3.4 Simple Operations

Parentheses are used to tell Python do this operation first. In addition, there is an operator precedence which are perform in the following order.

- `**`
- `*`, `/`, and `//`
- `+` and `-`

Operators at the same level have the same precedece, so when having multiple operators of the same level they are evaluated from left to right.

```
In [1]: 3 + 2 * 2
Out[1]: 7

In [2]: 3 * 10 / 2
Out[2]: 15.0

In [3]: 2 + 1 - 10
Out[3]: -7

In [4]: 9 // 2 * 4
Out[4]: 16

In [5]: 4 + 7 % 3
Out[5]: 5

In [6]: 5 * (1 + 3)
Out[6]: 20
```

6 Variables

6.0.1 Binding variables and values

In Python, we do an assignment using an equal sign.

```
x = 1.234567
```

In the previous piece of code, a variable `x` is being assigned a value 1.234567. This statement is basically binding or associating with the name `x` the value 1.234567, and now it is possible to use `x` wherever one would want by typing `x`.

6.1 Abstracting Expressions

Values of expressions are given a name for several reasosn. For example, one can reuse the name without redoing the computation to obtain the value, or it makes code easier to understand. In adittion, it makes code easier to change later.

```
pi = 3.141592
radius = 2.2
area = pi * radius ** 2
```

6.2 Programming vs. Math

In programming expressions are not like math expressions, i.e., one does not solve for x. In Python, one can add a comment, a line that is completely ignored by the interpreter, using the number sign or hash `#`.

```
# This is a comment
pi = 3.141592
radius = 1
# area of a circle
area = pi * radius ** 2
radius = radius + 1 # this is the same as radius += 1
```

An assigment statement find the value on the right hand side of the expression, then take the name on the left and assign that name to that value.

6.2.1 Changing Bindings

It is possible to rebind a variable name in a new assignment statement. The previous value may still be around in memory, but it's lost, i.e., there is no way to get to it.

```
pi = 3.1415
radius = 2
area = pi * radius ** 2
radius += 1
```

In the previos suite of code, a suite of code the Pythonic way to call a block of code, the value for `area` does not change until one tells the computer to do the calculation again.

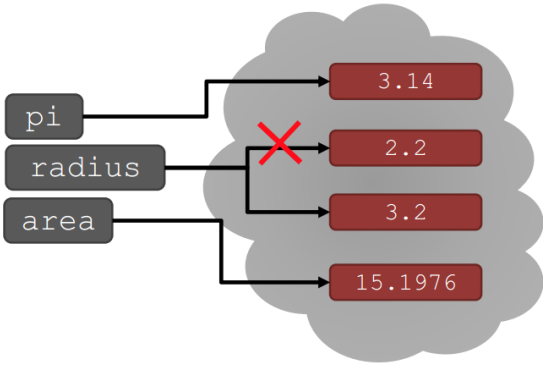

Figure 2: Changing a variable value

7 Operators and Branching

7.1 Comparison Operators on `int` and `float`

If `i` and `j` are either a `float` or an `int`, the following comparison operators can be applied,

- `i > j` it is evaluated to `True` if `i` is greater than `j`.
- `i >= j` it is evaluated to `True` if `i` is greater than or equal to `j`.
- `i < j` it is evaluated to `True` if `i` is less than `j`.
- `i <= j` it is evaluated to `True` if `i` is less than or equal to `j`.
- `i == j` it is evaluated to `True` if `i` is equal to `j`.
- `i != j` it is evaluated to `True` if `i` is not equal to `j`.

7.2 Logic Operators on `bools`

If `a` and `b` are any variable names:

- `not a` $\longrightarrow$ `True` if `a` is `False` or `False` if `a` is `True`.
- `a and b` $\longrightarrow$ `True` if both are `True`.
- `a or b` $\longrightarrow$ `True` if either or both are `True`.

7.3 Branching Programs

The simplest branching statement is a *conditional*:

- A test which is an expression that evalues to either `True` or `False`
- A block of code to execute if the test is `True`.
- A block of code to execute if the test is `False`.

In Python, it is not necessary to have the `False` block. It is mandatory to have the `True` block.

```
x = int(input('Enter an integer: '))

if x % 2 == 0:
    print('Even')
else:
    print('Odd')
print('Done with conditional')
```

In Python, indentation is very important not only because it tells which pieces of code are associated together, but also because not providing indentation will generate an error. This is different in other programming languages such as C/C++ or Java in which indentation is only used for readability purposes.

7.3.1 Nested Conditionals

It is possible to have nested conditionals, i.e., a conditional inside another one.

```
if x % 2 == 0:
    if x % 3 == 0:
        print('Divisible by 2 and 3')
    else:
        print('Divisible by 2 and not by 3')
elif x % 3 == 0:
    print('Divisible by 3 and not by 2')
```

7.4 Compound Booleans

As it has been previously mentioned, it is possible to combine booleans,

```
if x < y and x < z:
    print('x is least')
elif y < z:
    print('y is least')
else:
   print('z is least')
```

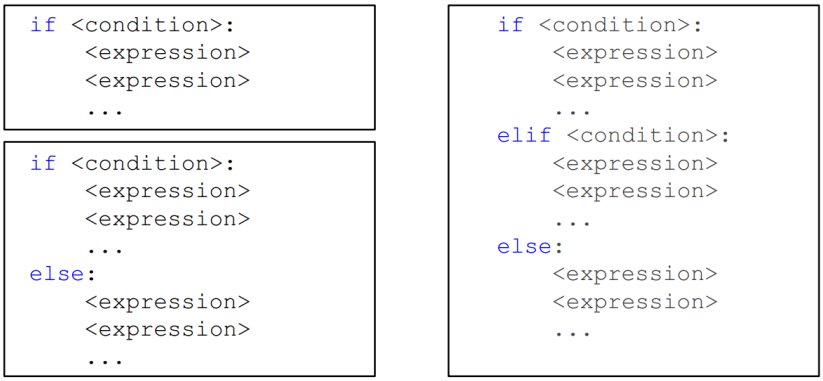

Figure 3: Control flow of branching

7.5 Control Flow - Branching

Basically in branching, a condition is evaluated and this value is either `True` or `False`, then a suite of code is evaluated depending of this value. Figure 3 shows the possible ways to use branching in Python.

To sum up, branching programs allow making choices and do different things, but each statement is executed at most once. This means that the maximum time to run the program depends only on the length of the program. As a consequence, these are the so-called linear programs because they run in constant time, i.e., each instruction is executed at most once. However, they do not provide the power to really build interesting algorithms.