6.00.1.x - Week 1 - Python Basics

September 14, 2020

1 Video 1: Introduction

What do we want you to take away?

How to think computationally, algorithmically. How can I describe this problem in order to get the computer to do it?

- Data structures.
- Iteration and recursions as computational metaphors.
- Abstraction of procedures and data types.
- Organize and modularize systems using object classes and methods.
- Different classes of algorithms, searching and sorting.
- Complexity of algorithms.

What does a computer do?

1. Perform calculations.
    - Built in to the language.
    - Ones that you define as the programmer.
2. Remember results.

Are there limits?

- Some problems are too complex.
- Some problems are fundamentally impossible to compute.

2 Video 2: Knowledge

Types of knowledge

1. Declarative knowledge: statements of fact.

   Square root of a number x is y suh that y*y = x

2. Imperative knowledge: is a recipe or "how-to".

   Recipe for deducing square root of number x.

What is a recipe? (aka algorithm!)

1. Sequence of simple steps.
2. Flow of control process that specifies when each step is executed.
3. A means of determining when to stop.

3 Video 3: Machines

How to capture a recipe in a mechanical process?

1. Fixed program computer.

   Examples: Hand-held computer, Alan Turing's Bombe.

2. Stored program computer.

   Can load a program and execute its instructions.

Basic Machine Architecture

1. Memory – Data, or instructions (a program).
2. Input / outputs.
3. ALU: Arithmetic Logic Unit.

Works between the IO and the memory. Performs primitive operations

4. Control Unit.

Has a program counter. It reads each instruction and then adds one (to move to the 2nd instruction, and so on). Eventually a test might be reached, and depending on its result, the counter will be changed.

Stored Program Computer

1. Sequence of instructions stored inside computer.

   Built from predefined set of primitive instructions: logic, tests, moving data.

2. A special program (interpreter) executes each instruction in order.

   Uses tests to change flow of control.

   Stops when done.

Stored Program Computer

- Turing showed you can compute anything using 6 primitives.

  Example: right, left, scan, read, write, do nothing (Turing Machine).

- Modern programming languages provide more convenient set of primitives.
- Anything that's computable in one (Turing Complete) language is computable in any other (Turing Complete) programming language.

4 Video 4: Languages

How to create recipes?

- A programming languages provides primitive operations.
- **Expressions** are complex but legal combinations of primitives in a programming language.
- Expressions and computations have **values** and **meanings** in a programming language.

Primitive constructs

- English: words.
- Programming Language: numbers, strings, simple operators.

Syntax: Is this a legal sentence or not?

- "hi"5, isn't syntactically valid.
- 3.2*5 is syntactically valid.

Semantics: What does this expression evaluate to?

- Static semantics: "Which syntactically valid strings actually have a meaning?"
  - 3 + "hi", syntactically valid (primitive, operator, primitive), yet not semantically valid.
- In a programming language, a both syntactically and static semantically correct expression will have only **one** meaning.

Type of errors

- Syntactic errors: easily caught.
- Static semantic errors: can cause unpredictable behaviour, some languages check for them.
- No semantic errors but different meaning than what was originally inteded: these are the worst errors.

5 Video 5: Types

Program: Sequence of definitions and commands. Definitions are evaluated. Commands are executed.

- A program will either be directly typed in a **shell** or stored in a **file** that is read into the shell and evaluated.

Objects: programs manipulate data objects.

- Objects have a type that defines the kinds of the things programs can do to them.
- Objects are **scalar** (cannot be subdivided) or **non-scalar** (have internal structure that can be accessed).

Scalar Objects

- int - Represent integers.
- float - Represente real numbers.
- bool - Represent Boolean values True and False.
- NoneType - Special and has one value, None.
- Can use type() to see the type of an object.

Type Conversions (Cast)

- Can converb object of one type to another.
  - float(3) converts integer 3 to float 3.0
  - int(3.9) trumps float 3.9 to int 3

Interesting example

- 3+5 as an input, gives an out: 5.
- print(3+5) gives no out, since no value returned, just something printed.

"Any expression that is syntactically valid has a value, which is itself a type."

Operators on ints and floats

- Sum, difference and product, between i and j:
  - If both are ints, result is int.
  - If either or both are floats, result is float.
- Division i/j: result is float.
- Int Division i//j: result is int, quotient without reminder.

- i%j: the remainder when i is divided by j.
- i**j: i to the power of j.

6 Video 6: Variables

Equal sign is an **asignment** of a value to a variable name. An assignment binds name to value.

Interesting tip

radius = radius + 1, can be replaced by, radius +=1 .

7 Video 7: Operators and Branching

Comparison operators on int and float

- i>j
- i>=j
- i < j
- i <= j
- i == j, equality test, True if i=j
  - Interesting point about comparisons. Python interpreter compares two numbers by **promotion** both to the more complex type. So for example in the comparison 10 == 10.0, the value of the expression is True, since both are promoted to float and are therefore equal.
- i != j, inequality test, True if i not equal to j

Logic operators on bools

- not a, True if a is False.
- a and b, True if both are True.
- a or b, True if either or both are True.

Branched programs

- Indentation is important: each indented set of expressions denotes a block of instructions. This indentation also provides a visual structure that reflects the semantic structure of the program.
- Branching programs allow us to make choices and do different things. But still the case that at most, each statement gets executed once.

8 Video 8: Bindings

Variables / Bindings

- Name
  - Descriptive, meaningful.
  - Helps you re-read code.
  - Cannot be keywords (int, float, etc.).
- Value
  - Information stored.
  - Can be updated.

Example Needing to create a "temp" variable in order to exchange the values between two variables. Can't just do it without it.

9 Video 9: Strings

Types of variables: int, float, bool, string!

String: letters, special characters, spaces, digits. Enclosed in "" or ''.

Difference with Matlab treatment of strings: Python will let you "sum" two strings in order to concatenate them (exactly like strcat would do in Matlab).

Concept of overloading an operation

- When using addition between strings, we say we are 'overloading' it because we are telling the operation to do something different to the arguments due to their type.

Operations on strings

- Succesive concatenation: 3*'eric' outputs 'ericericeric'
  - This is yet another example of overloading an operation.
- Length: len('eric') outputs 4
- Indexing out of an object: 'eric'[1] outputs 'r'. Counting starts at zero. 'eric'[0] will output 'e'.
- Slice a string apart:
  - 'eric'[1:3] will output 'ri'
  - 'eric'[:3] will output 'eri'
  - 'eric'[1:] will output 'ric'
  - 'eric'[:] will output 'eric'

Other operators

- 'in' operator tests for collection membership (a 'collection' refers to a string, list, tuple or dictionary).
  - element in coll, evalues to True if elemenet is a member of the collection coll, and False otherwise.
- 'not in' is exactly the same but the value is the opposite.

Advanced String Slicing

- You can slice a string such a s[i:j], which gives you a portion of string s from index i to index j-1.
- You can also slice a string like this s[i:j:k], which gives a slice of the string s from index i to index j-1 with step size k.
  - A cool idea in this area is using the following, s[::-1], to print string s in its entirety but backwards!

10 Video 10: Input/Output

Print: used to **output** stuff to console.

[1]:
```
x = 1
print(x)
```

```
1
```

[2]:
```
x_str = str(x)
print("my fav num is", x, ".", "x = ", x)
```

```
my fav num is 1 . x =  1
```

Print is printing each element followed by a space.

[3]:
```
print("my fav num is " + x_str + ". " + "x = " + x_str)
```

```
my fav num is 1. x = 1
```

Printing a concatenation of strings, letting me control the spaces more directly. Of course the possibility of choosing a separator works.

Input

- Input expects everything to be a string.

[4]:
```
text = input("type something'")
```

```
type something'7
```

[5]:
```
text
```

[5]:
```
'7'
```

The double quotes are there because I typed them in. Input expects everything to be a string, so if I don't want the quote marks, I don't need to add them there. But what happens if I want to input a number (and have it saved as type = int, for example)? I need to cast it.

[6]:
```
num = int(input("Type a number..."))
```

```
Type a number...4
```

[7]:
```
print(5*num)
```

```
20
```

11 Video 11: IDEs

- Integrated development enviroment (IDE) comes with:
  - Text editor.
  - Shell.
  - Integrated debugger.

12 Video 12: Control Flow

Branching program: Software that has at least two branches of different actions that will (or not) happen depending on the result of a test.

Up until now, with only if (and elif and else) statements, there is no way to implement an infinite 'loop' withot using infinite lines of code. Therefore, while loops are introduced.

While loops

- **condition** evaluates to a Boolean.
- If **condition** is True, do all the steps inside the while code block.
- Check **condition** again.
- Repeat until **condition** is False.

One can see that while loops can have some inconvenients, and that is what gives place to a more convenient kind of loop, called the **for loop**.

For loops

- Each time through the loop, **variable** takes a value.
- First time, **variable** starts at the smallest value.
- Next time, **variable** gets the prev value +1.

[8]:
```
for n in range(5):
    print(n)
```

```
0
1
2
3
4
```

- A useful tool within for loops, is the keyword **break**. It will stop the execution of the loop at that point, breaking out of it.

For loops vs while loops

- For loops have a known number of iterations, while loops have an unbounded number of iterations.
- Both loops can end early via **break**.
- For loops use a counter, while loops may use a counter.
- Can always rewrite a for loop using a while loop. May not be able to rewrite a while loop using a for loop.

Useful nomenclature for loops

a += b is equivalent to a = a + b

a -= b is equivalent to a = a - b

a = *b is equivalent to a = a* b

a /= b is equivalent to a = a / b

13 Video 13: Iteration

The same code can be used multiple times. First run a test, then run the loop body and keep running the test until the test evaluates to False. Some properties of iteration loops:

- Need to set an iteration variable outside the loop.
- Need to test variable to determine when done.
- Need to change variable within the loop, in addition to other work.

Information about the 'range' built-in function

- Form 1: range(stop) - 0,1,…,stop
- Form 2: range(start,stop) - start,start+1,…,stop
- Form 3: range(start, stop, stepsize). Ie: range(2,10,2) = 2,4,6,8.

14 Video 14: Guess and check

Guess and check methods are very useful, yet not always the most efficient solution to a given problem. It is based on a systematic approach at generating guesses to the answer of the problem.

- First version of the guess and check method. Created an iterative proposer of possible cubic roots of a given integer, by adding 1, trying whether it was or not the cubic root, and so on. Worked (for integers only), but slow and inefficient.

Useful to think about a **decrementing function**:

- Maps set of program variables into an integer.
- When loop is entered, value is non-negative.
- When value is <=0, loop terminates, and
- Value is decreased every time through loop

What can go wrong within a loop?

- The variable is not initialized. We are likely to get a **NameError**.
- The variable is not changed inside the loop. We get an **infinite loop** because it never reaches what is referred to as a **Terminating condition**.

So a **guess and check** algorithm:

- You are able to guess a value for solution.
- You are able to check if the solution is correct.
- Keep guessing until find solution or guessed all values.
- The process is exhaustive enumeration.

- **Second version** of the guess and check algorithm is created using a for loop.