5. Programming basics LING 471

PART 1

Assignment 1

- Due today! 11:59 PM!
- Any late submissions will be handled according to the policy laid out in the syllabus
  - Late penalty is automatically applied through Canvas
  - Portal closes 2 days after submission deadline (granting automatic 0s for any unsubmitted assignment)
  - A 0.5% penalty is applied for every late hour
  - Accommodations and homework extensions will be handled automatically, unless arranged separately

Assignment 2

- Released today
- Git skills used:
  - Cloning, staging, committing, pushing
- VSCode skills used:
  - Debugging, adding files to project
- Programming skills used:
  - Text processing, looping, I/O, etc. (a lot of them!)

Learning outcomes

- Define an expression and a statement
- Assign a value to a variable
- Create and manipulate lists and strings
- Write and evaluate Boolean expressions in Python
- Use conditionals as needed in programs

Some words of caution

- Until now, you have probably been able to coast and choose not to do the in-class activities or readings
  - If you have never programmed before, this will no longer be viable
  - You **must** practice using these concepts because you cannot just **intuit** them the night before
- Consider writing your own programs or working through some of the ones in the book if you need additional practice

Some words of encouragement

- Starting today, the material will seem very technical and precise
  - It is!
- But I promise you can do it!
- Programming is mostly just giving a list of very literal instructions
  - The struggle is learning how to express yourself to the computer
- It can feel frustrating if your code does not work or does something different than you intended
  - But this is a natural part of learning, and it is okay to not be good at something when you first start it! 🙂

Variables, expressions, and statements

Variables

- Sort of like a variable in algebra (and sort of not)
- Using some sort of **alphanumeric name** to stand in for a value
- Useful because typing out the full values EVERY time is tedious and liable to produce bugs
  - Imagine typing out π to 10 digits every time you wanted to use it! (Yuck!)

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897932
```

Assignment

- **Assignment** is the **action** of relating a value to a variable name
- In Python, it requires a left-hand side (LHS), "=" **operator**, and a right-hand side (RHS)
- LHS is a variable name
- RHS must **evaluate** to a value

Construction: LHS = RHS

- LHS – the variable name
- = – operator indicating assignment
- RHS – the value to assign to the variable name
- E.g.,
  - name = "Siyu"
  - class_num = 471

A note on operators

- Operators are special characters or word used to indicate a function
- **Binary** operators: require two arguments and often go between them, e.g., “+”, “-”, “=”
- **Unary** operators: require one argument, e.g., “not”, “-” (as a negative sign)
- Allow us to express some functions through syntax instead of functions
  - Compare `3+2` to `+(3, 2)`, `plus(3, 2)`, `3.plus(2)`, etc.

Variable names

- Each language has certain restrictions on what kinds of variable names you can use
- Python allows numbers, letters, and underscores in names
- Can’t have a number as the first character or use one of the keywords (like “if”) as a variable name
- General Python conventions
  - Variables start with lowercase letters
  - Indicate spaces with “_” (also called snake_case) and not camelCase

Choosing a variable name

- This is an art and not a science
- Generally, balance length of name with descriptiveness
  - You don't want every variable in your code to be "a", "b", "c"… because you won't know what the variables are
  - But you also don't want lots of variables like "list_before_adding_second_number_but_after_squaring_and_summing" either
- Rule of thumb: the more "interior" the variable, the shorter its name can be
  - Interiority is indicated with indentation in Python

Expressions

- Some combination of values, variables, and operators, including these by themselves
- Examples (from book)
  - 42
  - n
  - n+25
- An expression **evaluates** to some kind of value

Statement

- A piece of code that does something more than just evaluating to a value
- Examples
  - Variable assignment; `a = 42`
  - Printing; `print('Hello world')`

Programming activity 1

- Write an expression for the following prose: “Three plus two minus ten all divided by fifteen”
  - On a new line, store the value of that expression in a variable called “x”
- Assign the string 'hello' to a variable called “var”
- Print the result of the following expression: “(3**2+4**2)**0.5” (without the quotation marks)

Lists and strings

Lists

- A **list** is a very generic **data structure**
- Holds some kind of data in a given order
- Indicated using [ ]
- Very conceptually similar to the ordinary idea of a numbered list
- Each element can be of any type, including but not limited to, int(eger)s, floats, strings, functions

List creation

- Lists are created with the following construction:
  - [ELEMS]
  - [- opening bracket indicating the start of a list
  - ELEMS – 0 or more expressions, separated by commas
  -] – closing bracket indicating the end of the list
- E.g.,
  - `a = [1, 2, 3]`

List indexing

- If you want to get something out of a list, you have to index it
- Indexing is performed using the following construction:
  - LIST[IDX]
- LIST – the list you are indexing
- [– opening bracket
- IDX – the index of the element you want
-] – closing bracket

```
a = [1, 2, 3]
a[0] # 1
a[1] # 2
a[2] # 3
```

0-based indexing

- You may have noticed on the last slide that the first element had an index of 0 and not 1
- This is called 0-based indexing and is very common in programming
- So, the first element of a list is at index 0, the second element at index 1, and so on
- You can also count backwards: index “-1” is the (first-to-) last element, index “-2” is the second-to-last-element, and so on

List slices

- You can select multiple elements at once in a list with this construction:
  - LIST[IDX_1:IDX_2]
  - LIST – the list being indexed
  - [– opening bracket
  - IDX_1 – the first index to select
  - : - separating colon
  - IDX_2 – the first index **not** to select
  -] – closing bracket
- Not necessarily need both of, "IDX_1" and "IDX_2"

```
a = [0, 1, 2, 3]
# Useful:
a[0:2] # [0, 1]
a[0:3] # [0, 1, 2]
a[1:3] # [1, 2]
a[1:] # [1, 2, 3]
a[:3] # [0, 1, 2]
# Harder to understand:
a[:-1] # [0, 1, 2]
a[:-2] # [0, 1]
```

Changing the value of a list element

- Lists are **mutable** (i.e., they can be changed)
- You can reassign the value of the element at a particular index by combining indexing and assignment

```
a = [1, 2, 3]
a[0] = 3 # a is now [3, 2, 3]
```

Strings

- A string is a series of characters
- Can be thought of as a list of characters (and in some programming languages like C, they are exactly this)
- Can be indexed and sliced like lists
- But, strings are **immutable**, unlike lists
  - You must make a new string instead of changing an existing one
- Indicated as follows: 'CHARS'
  - ' – opening straight quote
  - CHARS – 0 or more characters to store in the string
  - ' – closing straight quote

Other common list and string operations

LISTS

- “+” – concatenation
  - [1] + [2] # [1, 2]
- “*” – replication
  - [1] * 3 # [1, 1, 1]
- “len()” – get length
  - len([1, 2, 3]) # 3

STRINGS

- “+” – concatenation
  - 'a' + 'b' # 'ab’
- “*” – replication
  - 'a' * 3 # 'aaa’
- “len()” – get length
  - len('abc') # 3

Programming activity 2

- Create a list that includes the integers from 1-9
  - Print the fourth element from that list
  - Print the fourth and fifth element from that list using slicing
  - Set the last element to 10
- Create a string of the integers from 1-9
  - Perform the same actions as for the list
  - You will have to make a new string using concatenation instead of setting the last element

Conditionals

Booleans

- An expression that is ultimately either true or false
- General binary operator construction:
  - BOOL_EXPR$_1$ BOOL_OP BOOL_EXPR$_2$
  - BOOL_EXPR: an expression that evaluates to true or false
  - OP: an operator that works on Boolean values or evaluates to a Boolean value
- E.g.,
  - `3 <= 2`
  - `x > 2 and x < 5`

Relational operators

- Operators that assess (in)equality of values
  - == - is equal, e.g., 3 == 3 is true, 3 == 2 is false
  - != - not equal, e.g., 3 != 2 is true, 3 != 3 is false
  - > - greater than, e.g., 3 > 2 is true, 2 > 3 is false
  - < - less than, e.g., 2 < 3 is true, 3 < 2 is false
  - >= - greater than or equal to, e.g., 3 >= 2 is true, 2 >= 3 is false
  - <= - less than or equal to, e.g., 2 <= 3 is true, 3 >= 2 is false

Logical operators

- Alter or join true and false values
  - and – true if both expressions are true
  - or – true if one or both expressions are true
  - not – inverts the truth value of an expression

Logical and

- `x and y`
- Returns true if both x and y are true

`2 < 3 and 5 > 2` -> true

`3 < 2 and 2 > 5` -> false

and		x	
		false	true
y	false	false	false
	true	false	true

Logical or

- `x or y`
- Evaluates to true if x is true, y is true, or both x and y

`3 > 2 or 3 > 4` -> true

`3 < 2 or 3 > 4` -> false

or		x	
		false	true
y	false	false	true
	true	true	true

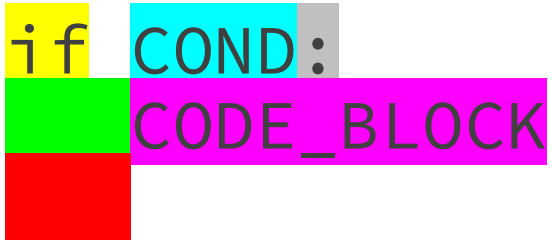
Logical not and de Morgan's Law

- `not x`
- Changes x from true to false or from false to true
- De Morgan's law involves negating logical and or logical or
  - `not (x or y) == (not x) and (not y)`
  - `not (x and y) == (not x) or (not y)`
  - May not be used that much in our code, but can be useful to know

Conditional execution

- Sometimes, we only want some code to run if a specific condition is met
- For example, maybe we want to print the sign (positive, negative, neutral) of a number
  - In plain language, a number is positive if it is greater than 0, negative if it is less than 0, or neutral of it is exactly 0
  - How do we do this in code, though?

if statements

- Conditional execution is run with if statements
- General construction
  - 

- if – keyword indicating conditional
- COND – the Boolean condition on which to run the following code
- : - indicator that some code is coming up
-  - beginning of indentation
- CODE_BLOCK – the code to execute
-  - end of indentation

Printing sign of number

- We've said that a number is positive if it is greater than 0, so we can write
  - ```
    if x > 0:
        print('+')
    ```
- But how do we check for negative or neutral signs?
    ```

Additional conditions

- We can use the elif keyword, short for “else if” to indicate an additional condition
- The construction is the same as for if, but elif is used instead
- So, we can then have
  - ```
    if x > 0:
        print('+')
    elif x < 0:
        print('-')
    ```
- But what about 0?
    ```

Alternative conditions

- One final keyword used with conditionals is “else”, which roughly means “for any other condition, do this”
- Else does not have a condition following it
- See the full code for printing the sign on the right

```
if x > 0:
    print('+')
elif x < 0:
    print('-')
else:
    print(0)
```

Programming activity 3

- See skeleton here
- Write a program that does the following:

1. Set a variable “x” to be a number
2. If “x” is divisible by 5, print “Fizz”
3. If “x” is divisible by 3, print “Buzz”
4. If “x” is divisible by 3 and 5, print “FizzBuzz”
5. If “x” is not divisible by either 3 or 5, do not print anything