Contents

Recall

A binding is a way to reference value by a name in such syntax as:

$$x \leftarrow 0$$

We can see from Eq. 0 that the *name* x is referencing the value of the *expression* 0. Recall that this binding forms a *statement*.

Python

What is Python, how do we use it? Python is the, if not one of the most popular and prevelant programming languages in the world. For the purposes of this class it will be the language we translate our pseudocode too. For example the way to create a binding or *variable* in pseudocode is defined by Eq. 0. However in Python the syntax for such a creation is as follows:

```
x = 0
```

This is a simple enough concept, we can use the *binding* (variable) x inside an expression if we desire. Or we could print it (lab assignment).

Collections

Recall the main four data types we talk about in this class:

1. Numbers: $\mathbb{Z}, \mathbb{R}$
2. Characters
3. Booleans
4. Collections

Here we will discuss collections and there various appearances. First the theory of collections can be traced back to **set theory**. The notation for a set is usually a collection of items, deliniated by the , symbol all inside the [] symbols. As discussed in the **nomenclature** below the [ ] symbols denote a set.

$$\mathbb{Z} \equiv [..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...] \tag{1}$$

We can see in Eq. 1 an example of the set of all integers, this is an infinite set which we wont deal with in finite space. With that said the notation and structure is exactly the same as some collections we will use in this class. The most common type of collection we will use in this class is called an *Array*. An array is a finite, homogeneous collection of elements. What does that word soup mean? It means that an array has a known amount of elements or size and that size is sub-infinite (finite). Homogeneous means that an array has only one type of thing in it, this will become clear below.

$$x \leftarrow [1, 2, 3, 4, 5]$$

An example of arrays in pseudocode could look something like Eq. 1 above.

```
x = [1,2,3,4,5]
```

The equivalent python will look similar, see the above code.

Why Collections?

The question now becomes what are arrays good for anyway? Besides the obvious use case of set theory examples, we can more easily group large quantities of items for computations. Imagine the type **character**, while a single English letter is useless, the sequence of letters forms a word and the sequence of words forms a sentence, etc. One can come to their own conclusions on the usefulness of such a construct. As such we should not overlook such a constructs usefulness, in particular the usefulness for complex data. Imagine attempting to write a word using single characters.

$$\begin{gathered} x0 \leftarrow 't' \\ x1 \leftarrow 'h' \\ x2 \leftarrow 'e' \\ \text{word} \leftarrow x0 * x1 * x2 \end{gathered} \tag{2}$$

$$\text{word} \leftarrow ['t','h','e'] \tag{3}$$

While we will almost never see either Eq. 2 or Eq. 3, we will use variants of these. Of course it is obvious the advantage of set notation in Eq. 3. The Python language has defined a type called **String** which is a predefined array of character for our convenience.

```
word = "the" # completely valid python code
```

This is cool but how do we use something inside this array? Arrays in programming languages tend to be indexable, meaning we can specify which thing we want to use.

$$x \leftarrow \frac{[5,4,3,2,1]}{x_0} + x_1 \rightarrow 5 + \frac{4}{x_3} \rightarrow 2 \tag{4}$$

In pseudocode this would look like Eq. 4, we start counting the indicies from zero instead of one. Fun fact: the mathmaticians do not agree with us on this and start counting from one so sometimes you will see people and languages count start at one. For the purposes of this class we will always start counting from zero. The same logic as Eq. 4 but in python can be seen in the code below:

```
x = [5,4,3,2,1]
print(x[0] + x[1]) # implying 5 + 4
print(x[3])
```

The python code has been given print statements as a way to show you the exact output of each indexing operation. We observe that Python (unlike the more math oriented pseudocode) utilizes the "[]" symbol to denote an indexing operation. The why of this is unforntunetly outside the scope of this class but in another class down the line you will learn why.

Iteration

What is iteration? Iteration is something like walking through each index of a set. Usually when we iterate over a full set we want to perform some action at each index.

$$\begin{gathered} \forall x \in [1,2,3,4,5] s.t \\ \sum \leftarrow \sum + x \end{gathered} \tag{5}$$

We observe in Eq. 5 that we can calculate the sum of a set easily with iteration. While the pseudocode is simple to understand syntactically, be sure to analyze this semantically. The essence of

iteration is in thinking of all collections as sequences with an order that can be traversed. We can only think this way because of how RAM works and how the arrays themselves are allocated in RAM.

Live Coding for Python

Python iteration

Python handles iteration is a multitude of ways all of which basically accomplish similar tasks while utilizing different logic. For the purposes of this section we will utilize *for* loops. A for loop is a type of logic similar to the theory of ∀, in that it utilizes a collection for iteration. In a *for* loop there exist two basic ways to iterate, over a **custom range** or over a **list**.

```
for i in range(0,100):
  print(i)
```

In the above code there we use the **range** function to create a custom **list** (set) from zero to one hundred counting by one.

```
x = [0,1,2,3,4]
for i in x:
  print(i)
```

In the above code we have used a python **list** that we previously constructed. In both the code blocks the python code will act the same. Utilizing the *for* keyword we describe an iterator **i** that will represent each index of our preferred set. This type of iteration is useful when we want to traverse an entire list, say to find the sum?

Functions

Functions are sections of reusable logic that can be called upon non-locally easily. What I mean by that is that a function is normally an set of operations that form a higher order operation.

$$\text{sum } l = \sum l \tag{6}$$

We observe that in Eq. 6 the function sum is defined as the simple sumation operation. This function can now be used elsewhere in our pseudocode in place of the summation symbol. This example is not necessary, but what if the function is more complex than a single predefined operation though?

$$\begin{aligned} &\text{euclideanDist } p_0, p_1 : (x_x, y_x) \leftarrow p_x = \\ &\quad a \leftarrow |x_1 - x_0| \\ &\quad b \leftarrow |y_1 - y_0| \\ &\quad c \leftarrow \sqrt{a^2 + b^2} \\ &\quad \text{return } c \end{aligned} \tag{7}$$

We observe that Eq. 7 is a complex enough equation that we benefit from making it into function. It should be clear now that functions are usually logical producers that can be reused with little effort. The theory on when and how to create function is complex, I have found the best strategy is to dissect everything into separate functions as much as possible. How do we create these things in python you might ask?

```
def sum(l):
    ret = 0
    for i in l:
        ret = ret + 1
```

The above code is equivalent to the pseudo code found in Eq. 6. In python we use the *def* keyword to denote a function, then we name it and place parenthesis containing any *parameters. Parameters* are the arguments that one can feed to a function for it to work properly. We can see that after the parameter list the colon symbol is used to denote the logics location beneath our declaration. The logic is of course a simple for loop with a temporary variable to contain the resulting values. *hint: There is a much shorter ways to form this function, extra credit for finding it, and bringing it to me.* For the more complex logic of course a more complex function is needed.

```
import math

def euclideanDist(p0, p1):
    (x0, y0) = p0
    (x1, y1) = p1
    a = abs(x1 - x0)
    b = abs(y1 - y0)
    c = math.sqrt(a**2 + b**2)
    return c
```

While this function is in the scope of the class it is rather complex for the beginning of the semester and as such I will be bring it back up later. Line by line breakdown:

- **import math**: This imports the math library from the python standard library[1].
- **def euclideanDist(p0, p1):**: this declares the function **euclideanDist** and shows that it takes two parameters **p0** & **p1**, finally the colon symbol **:** directs us to the logic below.
- **(x0, y0) = p0**: This implies to python that **p0** is a tuple of the type **(int, int)** and extracts those values into **x0** & **y0** respectively.
- **a = abs(x1 - x0)**: This utilizes the absolute value function **abs()** in python, denoting that we want this result to be positive[2].
- **c = math.sqrt(a2 + b2)**: This is utilizing that library we imported earlier to invoke the square root function **sqrt**. The ** symbols here represent the *exponent* operation.

[1] later in the class we will come back to the python standard library
[2] recall that distance is always positive

Nomenclature

Σ l - is the *sumation* of the set l, this means that it takes each element of all and preforms the addition operation on them sequentially

$s.t$ or : - read “such that”

$\forall$ x $\in \mathbb{Z}$ - read “for all x in $\mathbb{Z}$”, denotes iteration over all indices of $\mathbb{Z}$

A $\longleftarrow$ B - denoting a binding where A will represent the value of B

A $\longrightarrow$ B - read A **implies** B, denoting the that the statement A being true implies that B is also true

[ ] - denoting a set

$\mathbb{Z}$ - set of all integers (real numbers or their negatives)

$\mathbb{R}$ - set of all real numbers (rational numbers/irrational numbers, including fractions and their negatives)

i $\in$ I - denotes that i is an element of the set I