Notes & Homework: June 12, 2021

How to talk about a list: indexes and values

Let's say you have a line of people. Can you draw some stick figures in a line for a bakery below? Let's pretend it's a Sunday and everyone is rushing to get some Croissants.

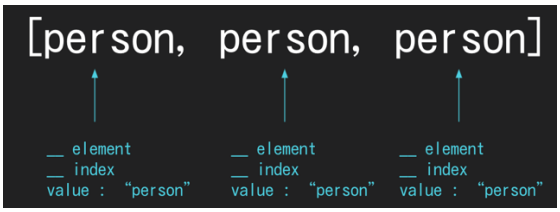
Now, looking at what you have above – how would you **refer** or **reference** each person's **position** in your list? Pause here and take a guess.

```
guess = ______
print(f'The guess was {guess}.')
#Output: The guess was ______.
```

You might have said first person in line, second person in line, and third person in line. The act of referencing position as "first", "second" or "third" is called **indexing**. But what if we had a list of bananas? Would "first person" be appropriate then? No! So, we need a convenient way of referring to things in a list. Instead of the word "things", I am going to use the world "element".

```
STOP! Run the first couple cells in notebook.ipynb for June 12. There's a fun game for this part!
```

Let's take a closer look at the list `["person", "person", "person"]` .



Each element's **position** in a list – that is "zeroth", "first", "second", etc. – is referred to as its **index** . What an element *is* is referred to as each element's **value** . In the above case, **value** would be "person". Each element of a list has both of these bits of information that we often refer to as the "index-value pair". You might've noticed earlier I wrote "zeroth". Python is **zero-indexed**, which means when we count our position in a list we start at 0.

Active solve: Let's assume I have a list `["a", "b", "c"]` and want to, for each **element** of the list, **print out** its index-value **pair**.

```
#Desired output:
Value: a, Index: 0
Value: b, Index: 1
Value: c, Index: 2
```

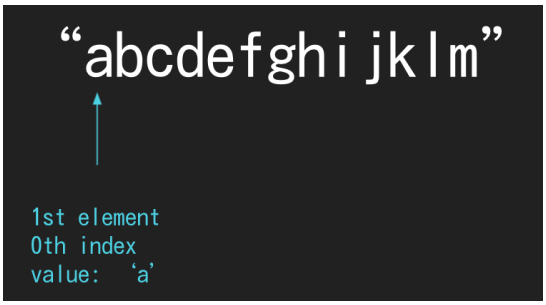
Can you write down your *thought process below in words first*, then try to code it out in `notebook.ipynb` ? Use the space below!

^Pro-tip: we call writing things out "in words" before coding **pseudo-code**.

Stuck? Check out the `hints.ipynb` file!

We just went through **indexing a list** , but we can also **index a string data-type**.

Remember: strings are made up of characters. If each **entry** in a list is referred to as its element, each **character** in a string is referred to as its element. Each character has an index, and a value.



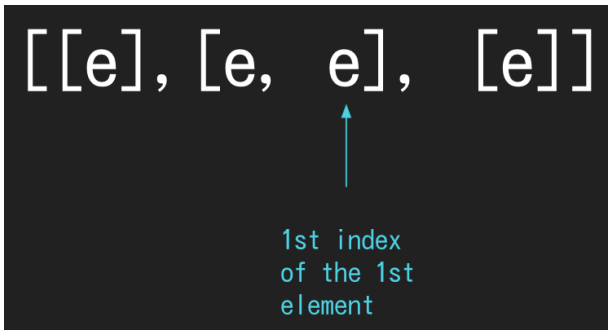
Active solve: Let's assume I have a string `"abc"` and want to, for each **element** of the string, **print out** its index-value **pair**.

```
#Desired output:
Value: a, Index: 0
Value: b, Index: 1
Value: c, Index: 2
```

Can you write down your *thought process below in words first*, then try to code it out in `notebook.ipynb` ? Use the space below!

Great job! We just went through indexing a **string**. Now what if we had a **list of lists**?

A **list of lists** is just lists inside of a list, and it's expressed by multiple **square brackets**.



Active solve, in your .ipynb: Let's assume I have a list of lists `[ [0, 0], [1, 1], [2, 2] ]` and want to, for each **element** of the list of lists, **print out** its index-value **pair**.

```
#Desired output:
Value: [0, 0], Index: 0
Value: [1, 1], Index: 1
Value: [2, 2], Index: 2
```

Hopefully you are getting pretty confident at this now. But what if we wanted to print it out like

```
#Desired output:
Value: 0, 0, Index: 0
Value: 1, 1, Index: 1
Value: 2, 2, Index: 2
```

If each list in our list of lists has **two** elements, in our for loop we can just increase the number of items we iterate over from 1 (like: `for i in list_of_lists`) to 2 like so:

```
l = [[0, 0], [1, 1], [2, 2]]

for i, j in l:
     print(i, j)

#Output:
0 0
1 1
2 2
```

What if I told you there was a quicker way to list out index-value pairs?

We can use a **built-in function** of Python. A "built-in function" (here's a list!) is something that's always available, like something that is always by your side and always in your toolbox when you're using Python. A fancy way of saying this is a function always available in the Python *interpreter*. We can use the built-in function **enumerate** to give us, or **return**, a **tuple** that has the structure (index, value) found in your list.

```
l = [1, 2, 3]

list(enumerate(l))

#Output: [(0, 1), (1, 2), (2, 3)]
```

Wait a second, we can use a for loop to **loop over** that thing, that `[(0, 1), (1, 2), (2, 3)]` !! That's why it's called an **enumerate object** that is **iterable**.

Active solve, in your .ipynb: Let's assume I have a list `['a', 'b', 'c']` and want to, for each **element** of the string, use **enumerate** to **print out** its index-value **pair** as a **tuple**.

Hint: take a look at the above, where `i[0]` gives us only the first element of the tuple.

```
#Desired output:
(a, 0)
(b, 1)
(c, 2)
```

Active solve: Let's assume I have a list `['a', 'b', 'c']` and want to, for each **element** of the string, use **enumerate** to **print out** *only the index*.

Hint: let's say we have a tuple `('a', 'b')` . `tuple[0]` will output `'a'` .

```
#Desired output:
0
1
2
```

Bonus question: Can you tell me why this error is happening?

```
[69]: l = ['a', 'b', 'c']
      for i in enumerate(l):
          i[0] = "Mwhahaha, I am changing this iteration's element's zeroth index!"
          print(i)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-69-d58116f6235c> in <module>
      1 l = ['a', 'b', 'c']
      2 for i in enumerate(l):
----> 3     i[0] = "Mwhahaha, I am changing this iteration's element's zeroth index!"
      4     print(i)

TypeError: 'tuple' object does not support item assignment
```

Space for your answer:

Stuck? Check out the `hints.ipynb` file!

Now, for the grand finale….

Active solve, in your .ipynb: Let's assume I have a list `['a', 'b', 'c']` and want to, for each **element** of the string, use **enumerate** to **print out** the index-value pair.

Hint: remember the `for i, j` syntax!

```
#Desired output:
Value: a, Index: 0
Value: b, Index: 1
Value: c, Index: 2
```

Math: `//` and `%`

I think you guys know how to add, minus, whatever; however, there are two symbols I want to explicitly go over for Python's way of doing arithmetic: `%` and `//` .

`//`

Before we go over this one, we need to understand the difference between a **float** and an **integer**.

So far, we've just called `1, 2, 3` **integers** but we have not yet dealt with numbers that are not whole numbers, that have decimal places, like `1.0, 2.1, 3.22` . These ones with decimals are called **floats**.

```
type(1)
type(1.0)

#Output:
<class 'int'>
<class 'float'>
```

Now, I think this one is easier if I just show you:

```
2 / 2
2 // 2

5 / 2
5 // 2

7 / 2
7 // 2


#Outputs:
1.0
1
2.5
2
3.5
3
```

`//` is called **floor division** and returns an **integer** instead of a **floating point** number (something with a decimal).

%

Let's review what a **remainder** is. When you divide 3 by 2, what's the remainder! 1, because 2 goes into 3 once and there's 1 left over. `%` gives you the remainder; for instance:

```
3 % 2
5 % 2
2 % 2
0 % 2
1 % 2

#Outputs:
1
1
0
0
1
```

The `%` is called "modulus", and we say `3 % 2` as "3 mod 2".

Active solve: Can you write the pseudo-code for checking if a number is odd using `%`?

Hint: odds are not divisible by 2.

```
#Output: what do you think this would output, if it was code?
```

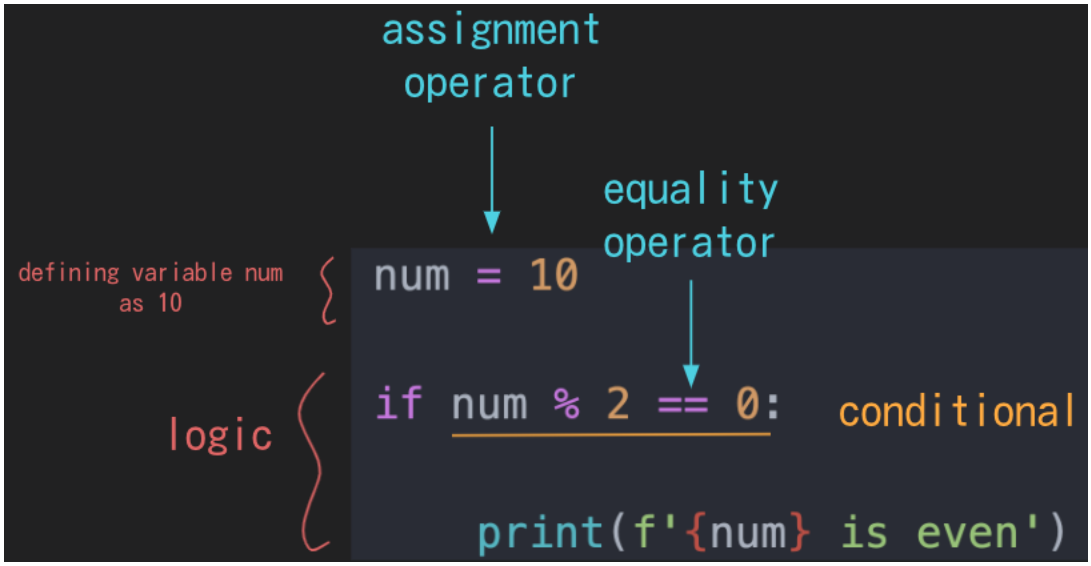
If statements: "conditional" executions

We've talked about how computers are written in base-two, or **binary** notation, 10010101. Even though life is not this way, computers think in terms of On/Off, True/False, 1/0 – in this vein, a computer's "logic" is just thinking about things as one way or the other.

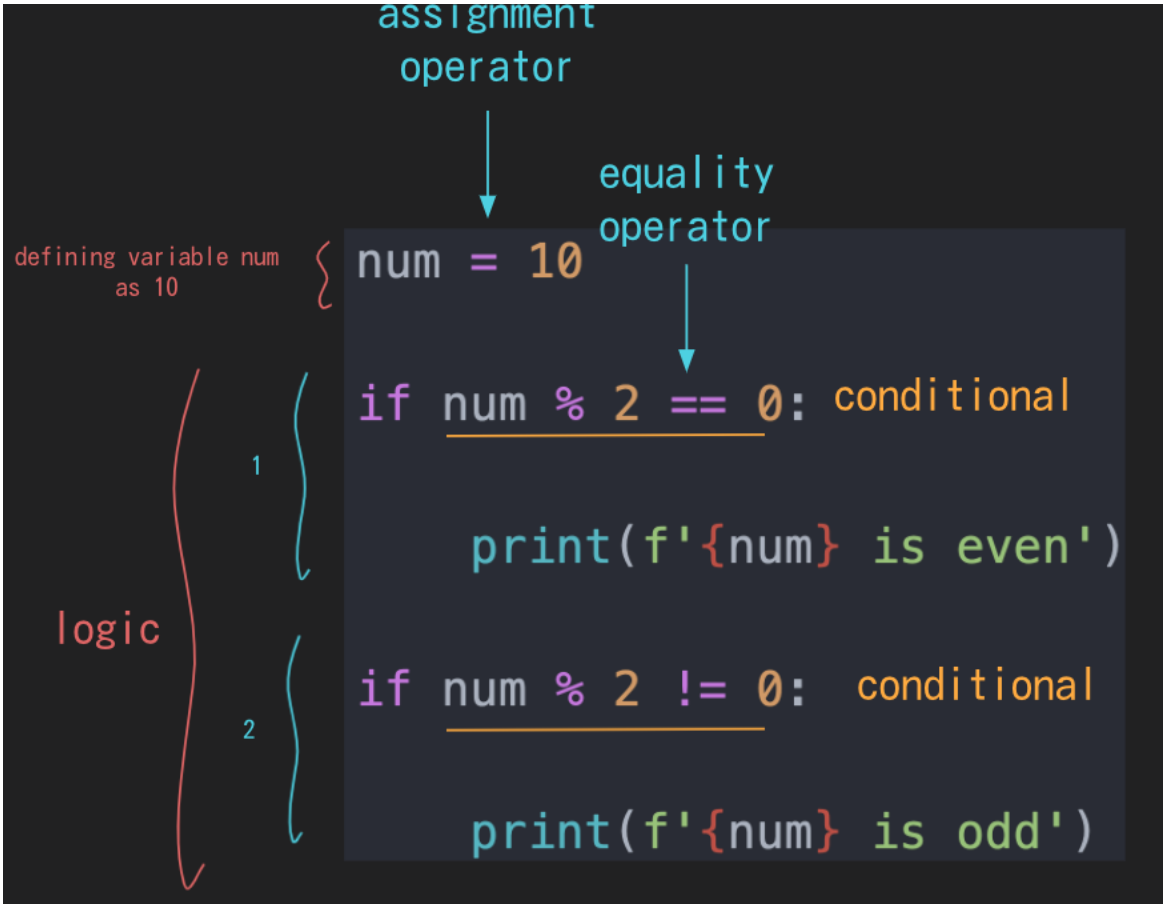
Taking your pseudo-code above and using this True/False idea, our program would output True or False.

Just as in English we say "if" followed by whatever we are supposing will happen, like "if it rains today", Python has a similar syntax: "`if`", followed by a "conditional" statement: "`if rain_today == True`".

Before, we were just using one equals sign, `=` , e.g. `l = [1, 2, 3]` . Now, we use `==` . Why? `=` is an **assignment** operator and `==` is an **equality** operator. The former assigns a variable a value, the latter checks if it's equal to something.
Putting it all together:



If we want to run, or "**execute**" multiple if statements as to check if a number is odd or even, we can just write them one after the other:



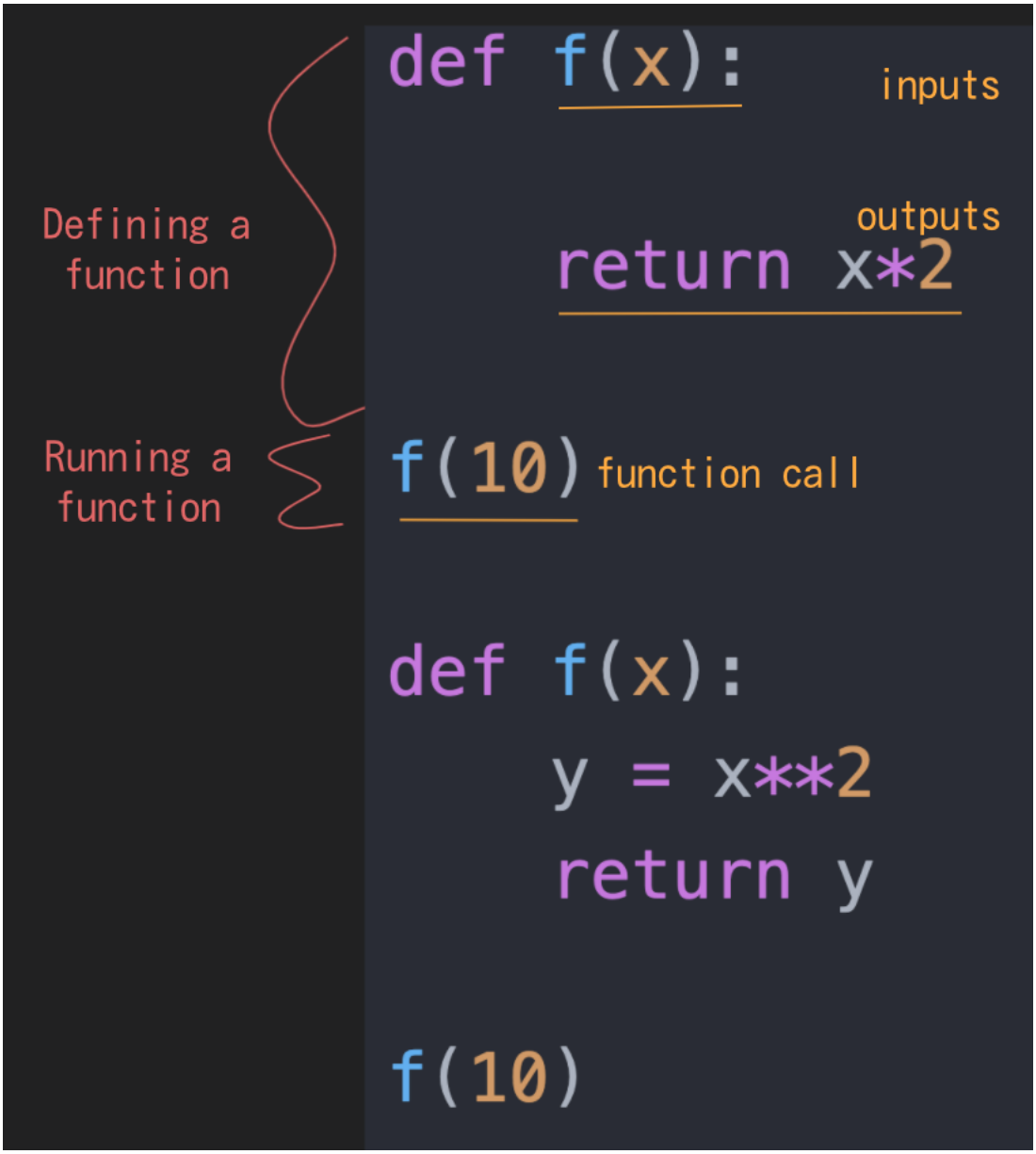
Active solve, in your .ipynb: Can you write some code that sums all even numbers in the range 1-100?

Hint: even numbers are divisible by 2.

Functions: inputs and outputs

We've been using the word "function" pretty liberally lately. We kind of intuitively get what a function is, right? Can you write down some characteristics you'd want out of a function?

A function is defined in Python by using the `def` keyword.



Homework Problems

Due: Thursday, Jun 17 12:00 AM

Please turn in via email to liamnisaacs@gmail.com . Please complete all the stuff on `noteshomework.ipynb` and submit your answers via a Jupyter notebook file. Solutions will be uploaded the morning of Friday, Jun 18, and you can expect feedback on your assignments Saturday, Jun 19.

Here's a brief list of what's on `noteshomework.ipynb` :

1. **All the "Active Solves"**

2. **Multiples of 3 and 5**

   If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

   Find the sum of all the multiples of 3 or 5 below 1000.

3. **Even Fibonacci Numbers**

   Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:

   1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

   By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.