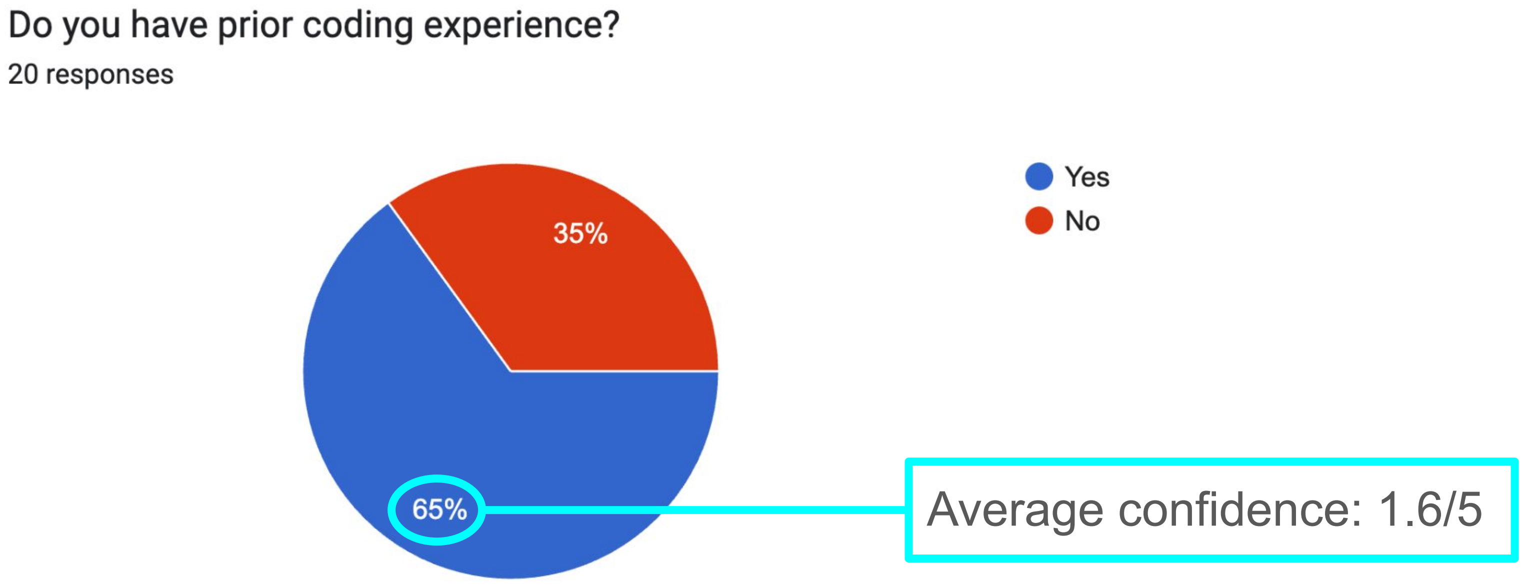
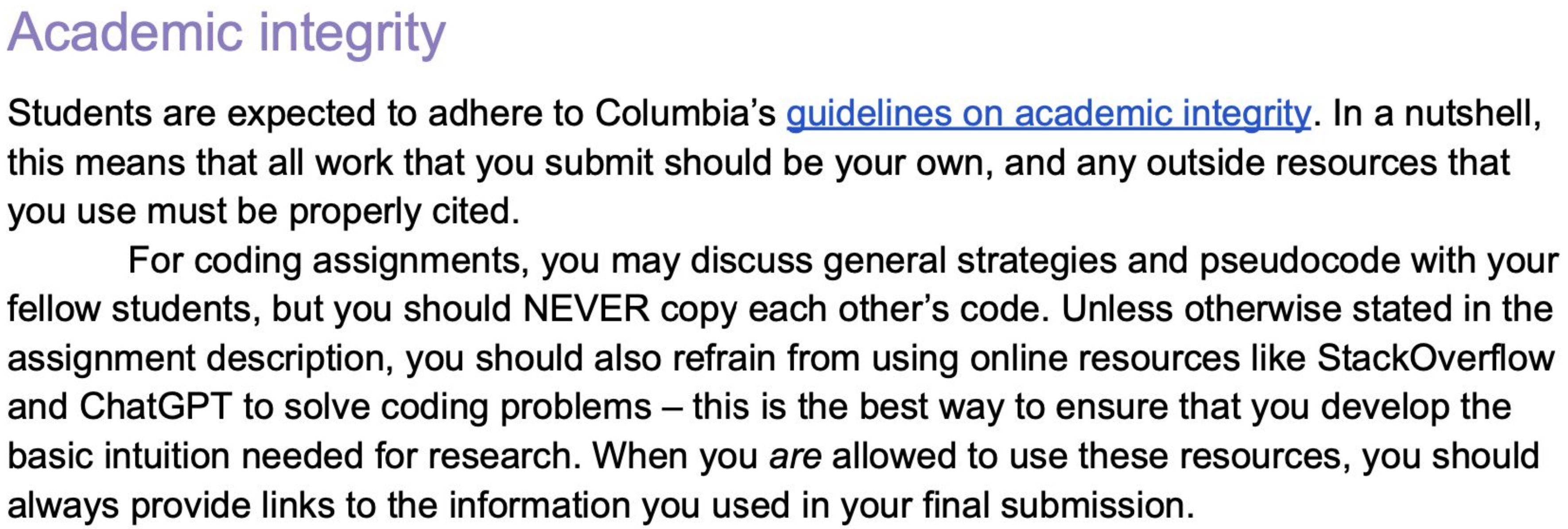
Introduction to coding

ASTR 2910 ⭑ Week 4

Registration form stats



Getting help from online tools



Academic integrity

Students are expected to adhere to Columbia's guidelines on academic integrity. In a nutshell, this means that all work that you submit should be your own, and any outside resources that you use must be properly cited.

For coding assignments, you may discuss general strategies and pseudocode with your fellow students, but you should NEVER copy each other's code. Unless otherwise stated in the assignment description, you should also refrain from using online resources like StackOverflow and ChatGPT to solve coding problems – this is the best way to ensure that you develop the basic intuition needed for research. When you *are* allowed to use these resources, you should always provide links to the information you used in your final submission.

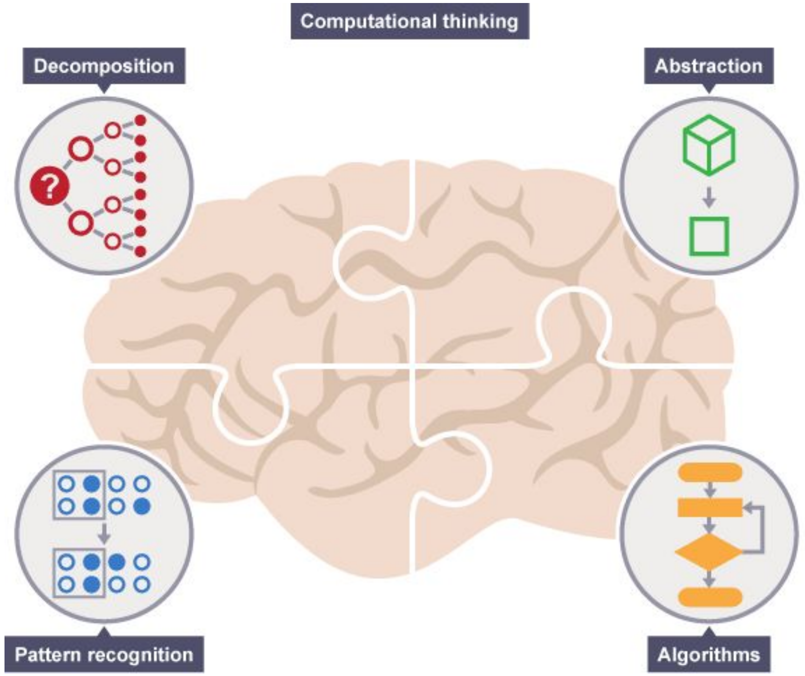
For most of today's class, we will *avoid* online tools.

Computational thinking

What does it mean to think computationally?

Coding = Writing step-by-step instructions for a task you want your computer to do

Breaking a large task into smaller parts



*Hiding irrelevant information for user-friendliness**

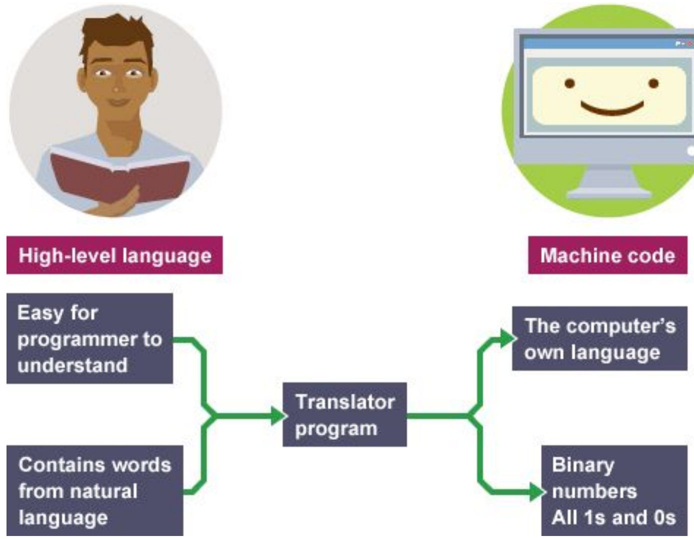
**Most relevant to designing packages*

Finding similar problems with known solutions

Simple steps or rules to solve each part of a task

How coding works

You write code (using a programming language) → The code is compiled/converted to machine code → The computer reads and executes the machine code


High-level language
Machine code
Easy for programmer to understand
Contains words from natural language
Translator program
The computer's own language
Binary numbers All 1s and 0s

Different programming languages have different "distances" to machine code.

Python is fairly high-level. Compilation and execution happen at the same time.

Terminal

What is the terminal?

“Old school” way of interacting with your computer and its file systems (e.g. creating, editing, and moving files).

A bit easier to use on Mac and Linux systems (which use Unix) than Windows.

Why not just use the GUI?

When connecting to remote servers to run expensive analyses or access data, you won’t usually have GUI access.

Tools to be aware of that we won’t use in this class:

- Working on remote servers: SSH (connecting), SCP/rsync (copying data)
- Command-line text editors: vim, emacs, nano

Common terminal commands

Unix	Windows	Explanation	Example
`~`	N/A	Points to home directory	
`ls`	`dir`	List files	`ls ~/Documents/folder`
`cd`		Change directory	`cd ~/Documents/folder/subfolder`
`pwd`	`cd`	Show path to current location	
`mkdir`		Make new directory	`mkdir new_dir`
`cp`		Copy file/folder	`cp code.py code_copy.py`
`mv`	`move`	Move file/folder	`mv code_copy.py new_dir`
`rm`	`del`	Delete file/folder	`rm new_dir/code_copy.py`

Absolute paths start with “/”; relative paths don’t

Python

Basic operations

```
1+1
4-5
3*4
7/2
```

Addition, subtraction, multiplication, and division use their usual signs. A double division sign (//) will perform integer division.

```
2**3
```

Exponentiation is denoted with **
(Fractional powers can be used to take roots.)

```
7%2
```

The modulo operator gives the remainder of a division operation.

```
[In [2]: 4/(5+1)**3
 Out[2]: 0.018518518518518517

[In [3]: (4/(5+1))**3
 Out[3]: 0.2962962962962962
```

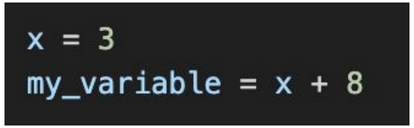
Parentheses enforce order of operations.

Variables

Variables are user-defined values that the computer remembers as long as your kernel is active (here: until we shut down iPython).

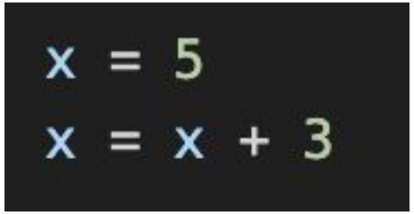
Variable names can contain letters, underscores, and numbers (as long as they aren't the first character).

- Valid names: `x`, `foo`, `my_variable`, `minutes10`, `_`
- Invalid names: `10minutes`, `my-variable`, `yay!`



```
x = 3
my_variable = x + 8
```

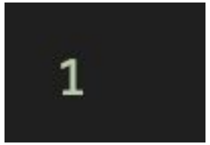
To define a variable: `name = value`
You can reference other variables in your definitions.



```
x = 5
x = x + 3
```

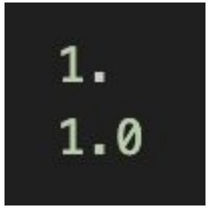
To update a variable, just redefine it. You can even redefine it with respect to itself!

Data types



```
1
```

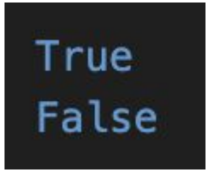
`int` – Integers; can be positive or negative



```
1.
1.0
```

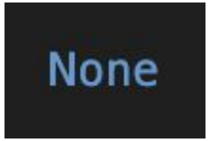
`float` – Numbers with decimal places; can be positive or negative

Your computer is only able to store floats up to a certain precision!



```
True
False
```

`bool` – Boolean; either true or false. Must be capitalized.



```
None
```

`NoneType` – Represents a lack of a value. Must be capitalized.

Data types

```
"Hello, world!"
'Hello, world!'
```

`str` – Strings of characters (text)

- Denoted with single or double quotes (to include a quote in the string, use `\'`)
- Special whitespace characters: `\n`, `\t`
- f-strings can be constructed using other variables:

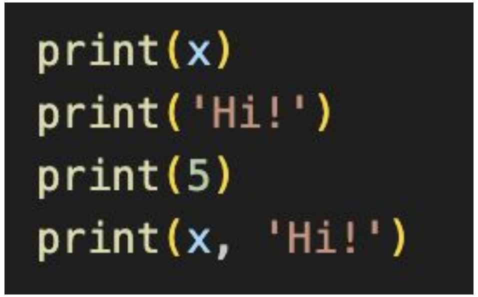
```
f'The value of x is: {x}'
```

- There are many built-in functions that can be used on strings:

```
name = 'Alex Masegian'
name.lower()
name.upper()
name.split()
name.startswith('A')
```

Printing

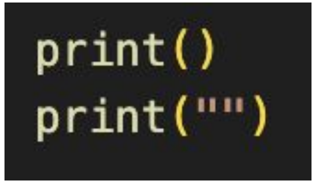
So far, we have been seeing our results as console output. But when we aren't working directly in the console itself, or we want to see more than just the result of the last line, we need to use `print()`.



```python
print(x)
print('Hi!')
print(5)
print(x, 'Hi!')
```

Most objects can be fed directly into `print()`

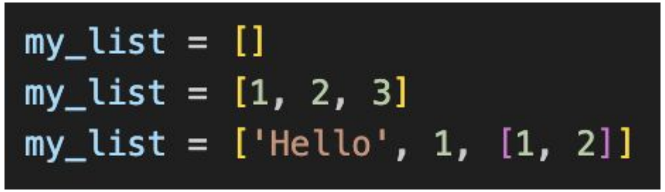
Objects separated by commas will be printed with a space in between them.



```python
print()
print("")
```

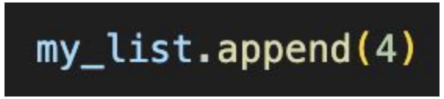
To print an empty line, simply leave the brackets empty, or fill in the empty string.

Data types



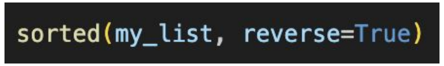
```
my_list = []
my_list = [1, 2, 3]
my_list = ['Hello', 1, [1, 2]]
```

`list` – Container for a collection of items. (Note that you can have lists inside lists!)



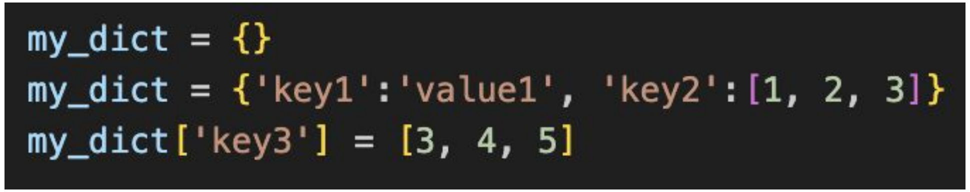
```
my_list.append(4)
```

Add to a list after it’s been defined.



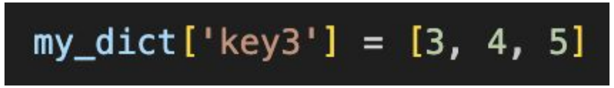
```
sorted(my_list, reverse=True)
```

Sort a list in ascending or descending (reverse) order.



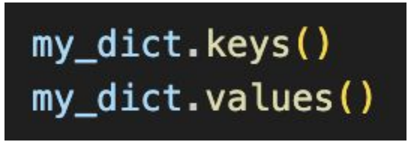
```
my_dict = {}
my_dict = {'key1':'value1', 'key2':[1, 2, 3]}
my_dict['key3'] = [3, 4, 5]
```

`dict` – Stores key-value pairs



```
my_dict['key3'] = [3, 4, 5]
```

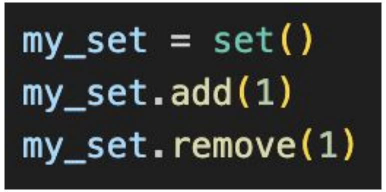
Add keys or update their values.



```
my_dict.keys()
my_dict.values()
```

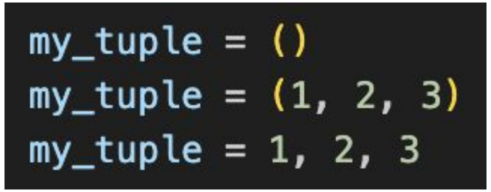
Get a list of all of the keys or all of the values.

Data types



```
my_set = set()
my_set.add(1)
my_set.remove(1)
```

`set` – Like a list, but only contains unique values



```
my_tuple = ()
my_tuple = (1, 2, 3)
my_tuple = 1, 2, 3
```

`tuple` – Like a list, but immutable

For strings, lists, dictionaries, tuples, and sets:

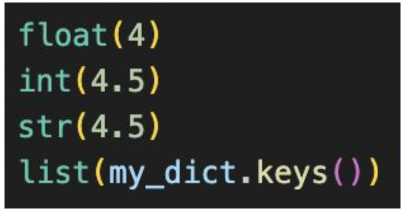
- Addition will concatenate
- You can check membership with the `in` keyword

Data types

Mutable types (lists, dictionaries, sets) can be changed in-place after creation.
Immutable types (strings, tuples) cannot be changed after creation.

You can test the type of a variable with the `type()` keyword.

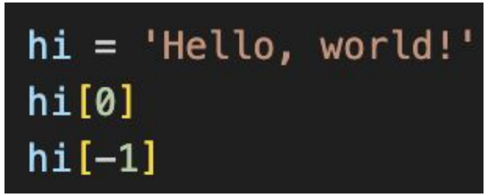
You can change a variable from one type to another by **casting**:



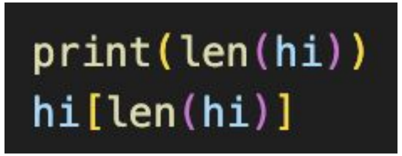
```
float(4)
int(4.5)
str(4.5)
list(my_dict.keys())
```

Indexing

Some data types, including lists and strings, can be indexed by position. Indices start at 0 and increase to `len(variable)`.

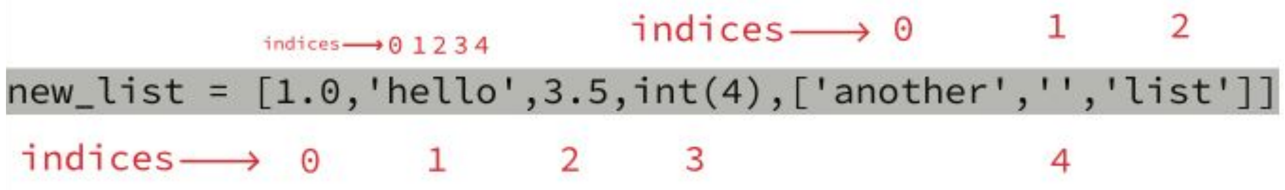


```python
hi = 'Hello, world!'
hi[0]
hi[-1]
```



```python
print(len(hi))
hi[len(hi)]
```

You can use multiple indices to navigate nested structures like this:



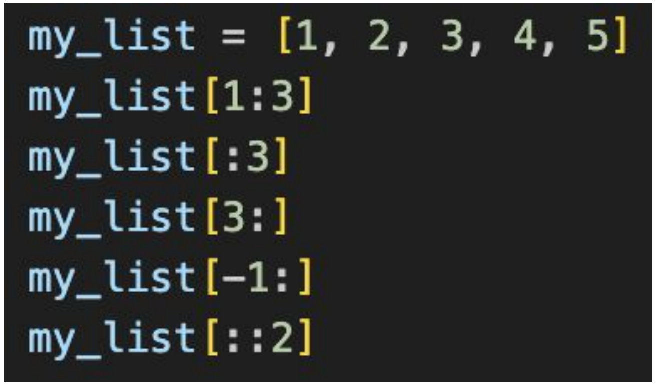
```python
new_list = [1.0,'hello',3.5,int(4),['another','','list']]
```

How would we access the letter “n” in “another”?

Slicing

Slicing allows you to select portions of lists or strings.

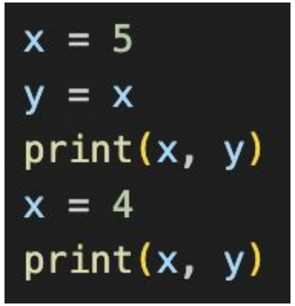
Python uses a closed-open indexing scheme: the starting index is included, but the ending index is not.



```
my_list = [1, 2, 3, 4, 5]
my_list[1:3]
my_list[:3]
my_list[3:]
my_list[-1:]
my_list[::2]
```

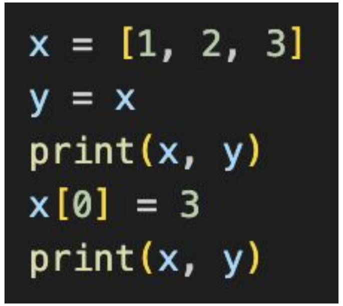
Copying variables

Strings and numbers are stored as discrete objects, but more complex structures (lists, dictionaries, etc) are stored as pointers, which affects copying.



```
x = 5
y = x
print(x, y)
x = 4
print(x, y)
```

Changing `x` will not change `y`, because each variable stores a unique copy of the number 5.



```
x = [1, 2, 3]
y = x
print(x, y)
x[0] = 3
print(x, y)
```

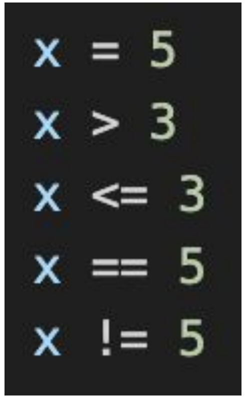
Changing `x` **will** change `y`, because both variables store the same pointer.

Reserved keywords

The names of data types and certain operations are **reserved keywords** with special definitions. If you overwrite them, you can’t access their built-in functions!

Never name a variable `list` or `str`!

Boolean operations

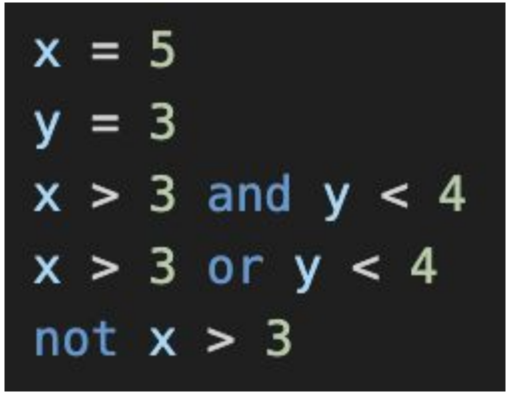


```
x = 5
x > 3
x <= 3
x == 5
x != 5
```

To compare values, use the typical signs (note the equals sign must always come second): >, >=, <, <=

== tests for equality (**must** be double!)

!= means “not equal”



```
x = 5
y = 3
x > 3 and y < 4
x > 3 or y < 4
not x > 3
```

To combine boolean operations, use `and` and `or`

To negate boolean operations, use `not`

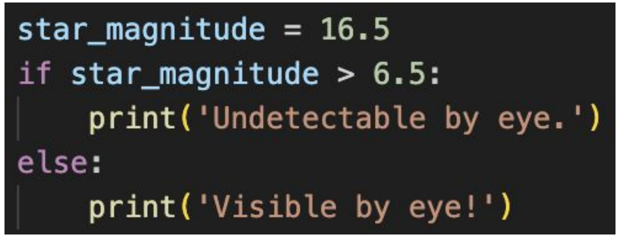
Parentheses can be used to signify order of operations.

Conditional statements

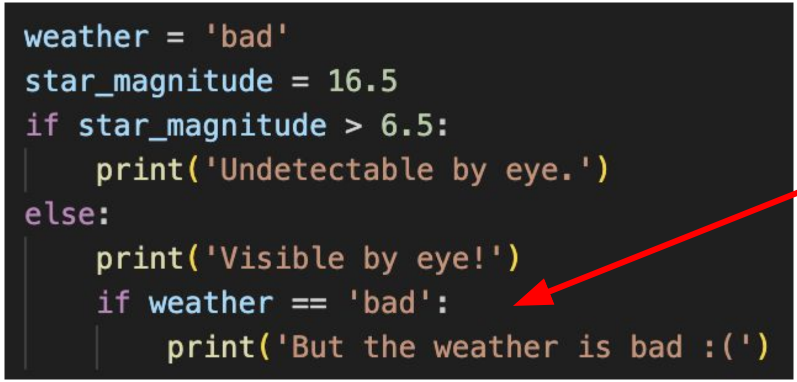
Used to create a "fork" in your code, where what happens next is determined by whether a certain statement evaluates to `True` or `False`.

Relevant reserved keywords: `if`, `else`, `elif`

Whitespace (indentation) and colons are used to denote the beginning and end of the conditional statement.



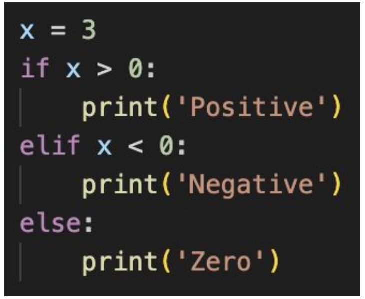
```python
star_magnitude = 16.5
if star_magnitude > 6.5:
    print('Undetectable by eye.')
else:
    print('Visible by eye!')
```



```python
weather = 'bad'
star_magnitude = 16.5
if star_magnitude > 6.5:
    print('Undetectable by eye.')
else:
    print('Visible by eye!')
    if weather == 'bad':
        print('But the weather is bad :(')
```

`if` statements can stand alone!

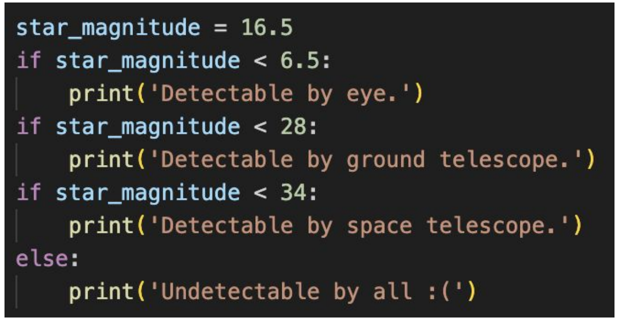
Conditional statements



```python
x = 3
if x > 0:
    print('Positive')
elif x < 0:
    print('Negative')
else:
    print('Zero')
```

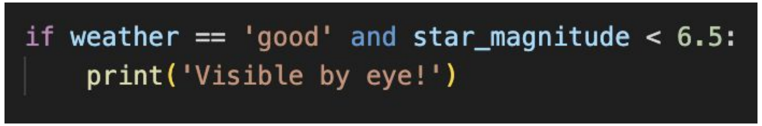
`elif` is used to add additional checks that only trigger when the previous checks fail. (Can be more than 1!)

Note that you don’t need to define the condition for `else`.



```python
star_magnitude = 16.5
if star_magnitude < 6.5:
    print('Detectable by eye.')
if star_magnitude < 28:
    print('Detectable by ground telescope.')
if star_magnitude < 34:
    print('Detectable by space telescope.')
else:
    print('Undetectable by all :(')
```

Sequential `if` statements will be triggered regardless of the outcome of previous checks.



```python
if weather == 'good' and star_magnitude < 6.5:
    print('Visible by eye!')
```

Conditionals can be made more complex by combining multiple boolean operations.

Exercises

1. What would each of the following boolean statements evaluate to?
   a. `True or False`
   b. `not False`
   c. `3.0 - 1.0 != 5.0 - 3.0`
   d. `3 > 4 or (2 < 3 and 9 > 10)`
2. Define a variable that stores an string of your choice. Using conditional statements, write a program that prints “vowel” if the string starts with a vowel and “consonant” otherwise.
3. Define a variable that stores an integer of your choice. Write a program that prints “fizz” if n is divisible by 3, “buzz” if n is divisible by 5, and “fizzbuzz” if n is divisible by 3 and 5.

Loops

Used to repeat code multiple times until a defined stopping condition is satisfied.

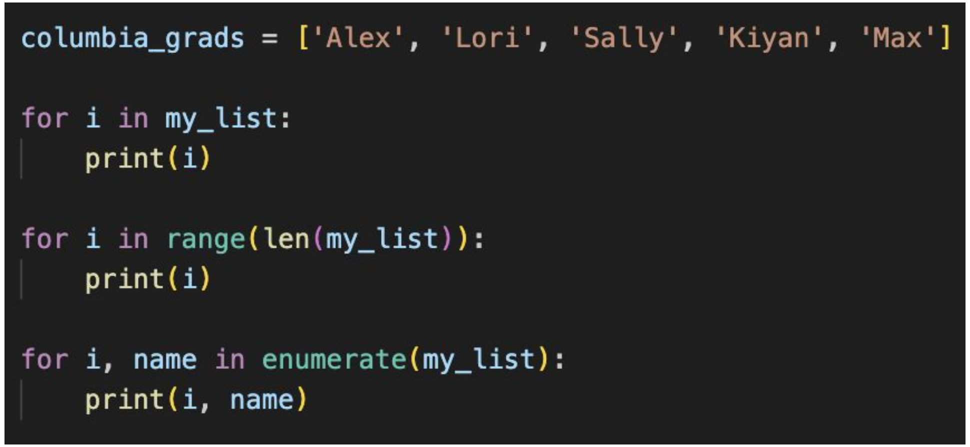
Certain data types (strings, lists, dictionaries) are iterable, meaning that their contents can easily be looped through.

Like conditionals, loops can be nested (e.g. iterating over a 2D array). You can also have conditionals inside loops.

Beware: loops are often the biggest culprit of slow code!

Loops

`for` loops extract each item from an iterable in order, terminating when the end of the iterable is reached.



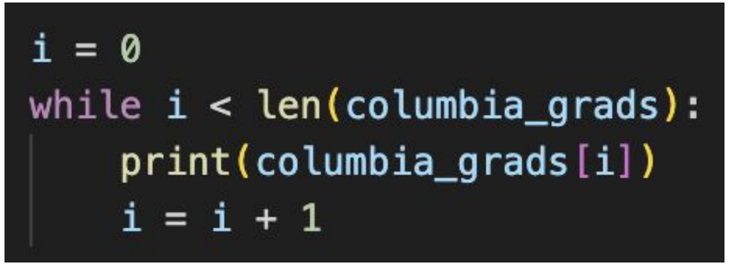
```
columbia_grads = ['Alex', 'Lori', 'Sally', 'Kiyan', 'Max']

for i in my_list:
    print(i)

for i in range(len(my_list)):
    print(i)

for i, name in enumerate(my_list):
    print(i, name)
```

`range(start, stop, step)` is useful for generating indices for iterables.
`enumerate(iterable)` allows you to iterate over both the indices and values.

Loops

`while` loops check a condition at the beginning of each iteration, terminating when that condition is no longer met.

Usually, you'll update one or more of the variables involved in the condition inside the loop so that a stopping condition is eventually reached.



```python
i = 0
while i < len(columbia_grads):
    print(columbia_grads[i])
    i = i + 1
```

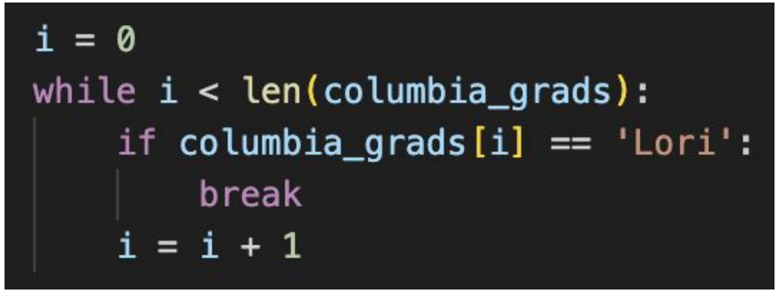
What would happen if the last line of code was removed?

`while` loops are most helpful for cases where you don't know how many times you'll have to loop (e.g. user input).

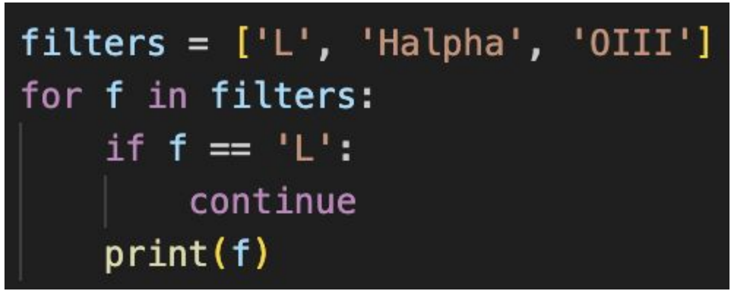
Loops

`break` will immediately terminate the current loop.

`continue` will skip to the next iteration.



```python
i = 0
while i < len(columbia_grads):
    if columbia_grads[i] == 'Lori':
        break
    i = i + 1
```



```python
filters = ['L', 'Halpha', 'OIII']
for f in filters:
    if f == 'L':
        continue
    print(f)
```

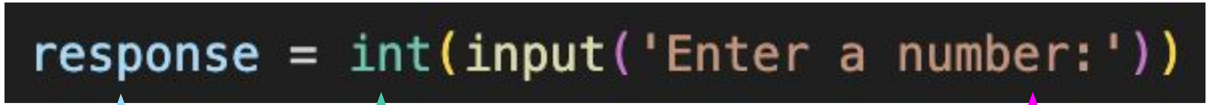
Loops

Common pitfalls:

1. Make sure that you're not adding to or deleting from the iterable you're looping over – this can cause unexpected behavior!
2. Accidentally made an endless `while` loop? Ctrl+C will stop execution.

User input

The `input()` command can be used to prompt the user for input. By default, whatever the user types will be returned as a string.



```python
response = int(input('Enter a number:'))
```

If we want our code to treat `response` as a number, we need to cast to `int` or `float`

Whatever you write here will be printed as a prompt for the user

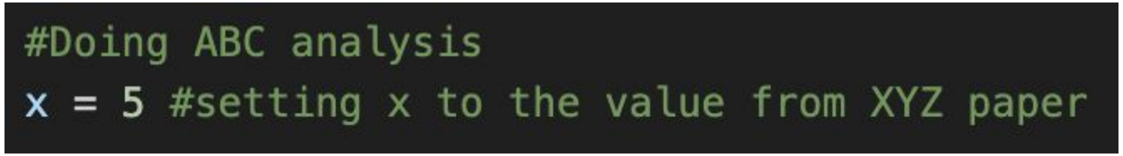
Make sure to store the input in a variable; otherwise it won't be saved!

Exercises

1. Write a loop that appends the numbers 0 through 9 to a list.
2. Define a variable containing a string of your choice. Write a program that counts up the number of vowels contained in the string.
3. Write a program that takes in user input and checks to see if the user inputted the word “quit.” If the user did not input “quit,” the program should print “try again!” and prompt the user for input again. The program should terminate if the user inputs “quit.”

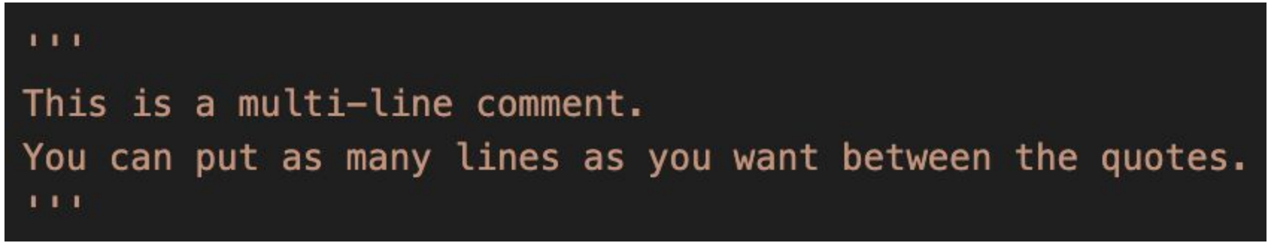
Comments

Single-line comments are denoted with #. Everything after the # will be a comment, even if there is also code on that line.



```
#Doing ABC analysis
x = 5 #setting x to the value from XYZ paper
```

Multi-line comments are denoted by three single quotes: ` ` `



```
'''
This is a multi-line comment.
You can put as many lines as you want between the quotes.
'''
```

Good coding practices

1. Use meaningful variable names (`x` is a terrible example)
2. Make use of white space to make your code as legible as possible
3. Leave comments (both for others and for yourself!)
4. Be consistent with whatever conventions you adopt (e.g. naming schemes)
5. Strive for efficiency (minimizing loops, etc.)

If you REALLY want to get into the weeds of what people consider “good style,” check out the PEP 8 guidelines.

Errors and debugging 🐛

Errors that Python catches: Caught during interpretation and will stop the execution of your code. An error traceback will be printed to the console (the last line of which is usually the most helpful).

Tips for debugging:

1. Don’t panic – read the error message!
2. If you’re really stuck, paste the error into Google or ChatGPT.

Errors and debugging 🐛

Silent errors: Your code still runs, but your output will be different than you expect… and it's up to you to figure out why.

Tips for debugging:

1. `print()` statements are your friend. Track your variables through your code.
2. Break your code into parts and figure out where the output stops being what you expect.