Variables

A **variable** is a name that you can assign a value or data to. Variables are useful because you can store data in them and use them later in calculations.

A variable is created using **a single equal sign**:

```
my_var1 = 5
```

The variable name goes on the left side of the equals sign, while the value or data goes on the right side.

WKSH EX1:	What is the name of the variable above? `my_var1`	What value or data have we assigned the variable above? `5`

You can name your variables whatever you want. However, note the following:

- Variables are **case-sensitive**.
  - `myvar` is not the same as `myVar`
- Your variable name **cannot start with a number**.
  - E.g., `2nd_number = 10`
- Your variable name **can only include letters, numbers, and underscores**.
- Your variable name **should not be the same as a built-in Python function**.
  - While this is not technically forbidden, you may overwrite important functions which you will then be unable to use later.
  - E.g., `list`, `int`, `str`

iPYNB EX1: Write the following code in the iPYNB EX1 cell in your Jupyter notebook.

- Create a variable named after your initials.
- Assign it a value equivalent to your birth year.

Data Types

In Python, every variable has a **type**. Different data types have different properties, and the type also determines what operations can be performed on that variable. Some common data types include:

Data Type	Description	Example
`int`	Whole numbers; can perform math operations on them	1, 2, 3, 10, 9999
`float`	Decimal numbers; can perform math operations on them	1.0, 1.00, 2.5, 3.1415, 100.
`bool`	Can take only take on values of `True` or `False`	`True`, `False`
`str`	Represents text, denoted using single or double quotation marks. You can concatenate strings using a + sign.	`'Hello world'`, `"20240213"`, `"avh_0213"`
`complex`	Complex numbers, with real and imaginary parts. The imaginary part is denoted with a `j`.	`1+2j`

You can use the built-in function `type()` to check the data type of variables you have created. This can be useful to check what sorts of operations can be performed on your variables.

For example, the following code…

```
X = 1234
Name = 'model_CESM'
print(type(X))
print(type(Name))
```

…would return:

```
int
str
```

You can **convert a variable to a different data type** simply by using its name as a function.

WKSH EX2: Given the following lines of code, what would the following variables be?

(A)

```
a = 3.141
a = int(a)
```

`a = 3`

(B)

```
b = 2
b = float(b)
```

`b = 2.0`

(C)

```
c = 2024
c = str(c)
```

`c = '2024'`

(D)

```
d = 0
d = bool(d)
```

`d = False`

Integers and Floats

Integers and **floats** represent **numbers**. They are most commonly used for math operations.

Operator	Name	Description
`a + b`	Addition	Sum of `a` and `b`
`a - b`	Subtraction	Difference of `a` and `b`
`a * b`	Multiplication	Product of `a` and `b`
`a / b`	True division	Quotient of `a` and `b`
`a // b`	Floor division	Quotient of `a` and `b`, removing fractional parts
`a % b`	Modulus	Integer remainder after division of `a` by `b`
`a ** b`	Exponentiation	`a` raised to the power of `b`
`-a`	Negation	The negative of `a`

iPYNB EX2: Write the following code in your Jupyter notebook.

- Create a variable called **x** that stores the value **3.1415**.
- Create a variable called **y** that stores the value of **4**.
- Create a variable called **z** that stores the **product of x and y**.

WKSH EX3: In **iPYNB EX2**, you created the variables **x**, **y**, and **z**. What **data type** will **x**, **y**, and **z** be? Hint: you can check the type of each variable using the `type()` function.

Variable	x	y	z
Data Type	float	int	float

WKSH EX4: What is the data type of the resulting variable when you add a `float` variable to an `int` variable? In your notebook, add together `a` and `b` from **WKSH EX2** and use the `type()` function to check.

Answer here: a float

Strings

Strings are used to represent **text**. You create a string by putting single quotes (`'text'`) or double quotes (`"text"`) around the text - both ways are equivalent.

You can use just about any characters on your keyboard in a string. For example:

```
name = 'alice v'
bday = '213'
pwd = "plsd0n'th@ckm3!"
```

Strings have different properties than integers and floats. You **can't do mathematical operations** on them, and **you can't add them to integers or floats**. However, they have special functions (called **methods**) you can use on them. The table below lists a few common ones, but you can find a longer list here: https://www.w3schools.com/python/python_ref_string.asp

Method	Syntax	Description
upper()	`myStr.upper()`	Capitalizes all characters in `myStr`
lower()	`myStr.lower()`	Lower cases all characters in `myStr`
replace()	`myStr.replace(old,new)`	In `myStr`, replaces the characters specified in `old` by the characters specified in `new`
split()	`myStr.split(dtr)`	Splits the characters of `myStr` by the delimiter specified by `dtr`

You can also **concatenate strings** using a `+`.
For example, the following code…

```
name+bday
```

…would output:

```
'alice v0213'
```

Check out the examples in the Jupyter notebook on how to use the above methods. Then, move on to the exercises below.

iPYNB EX3: Write the following code in the iPYNB EX3 cell in your Jupyter notebook.

Using the variables `name = 'alice v'` and `last = 'hsu'`, we want to create a new variable called `fullname` that is equal to `'Alice V Hsu'`.

1. First, **concatenate** the first and last names. Don't forget to add a space after the `v` in `name`.
2. **Click the link** to the online table of string methods (the link is in the Jupyter notebook).
3. Using the table, **find the string method** that is used to capitalize every word in a string.
4. **Read the documentation** on this method and then **use it** to capitalize your string.

Remember that different data types may not be compatible with each other. You can only perform **certain operations** on **certain data types**.

WKSH EX5: Why doesn't the following line of code work? (Put the code into a cell in your Jupyter notebook and see what happens when you try to run it).

```
mycalc = '4' + 3
```

Why does this happen? *Hint: error messages, while annoying, can be quite useful for debugging code!*

Explanation here: You can't add a string to an integer (or float).

What do you have to do first to the above line of code for it to work? Write the answer here, then implement this change in your notebook.

Answer here: `my_calc = int(4)+3` or `my_calc = float(4)+3` or `my_calc = '4'+str(3)`

Now, putting this concept into practice:

iPYNB EX5: Write the following code in the iPYNB EX5 cell in your Jupyter notebook.

A. Using the variables below, **create a string called `filename`** that includes all the components separated by a dash (-). `filename` should output `'MetMod-3-USA-19970213.csv'`.

```
model = 'MetMod'
  model_no = 3
country = 'USA'
date = 19970213
filetype = '.csv'
```

B. Now say we wanted to **split `filename` back into its component parts**. Choose the appropriate method from the table above and then write the code in the cell.

Challenge Warm Up: Temperature Converter

Concepts covered

- Basic maths
- Strings and string methods
- Data type conversion

Instructions

In this exercise, you will write some code that converts a temperature from Celsius to Fahrenheit and outputs it nicely as a string. The formula for converting a temperature from Celsius to Fahrenheit is:

$$F = C \cdot \frac{9}{5} + 32$$

Pseudocode

- **Create a variable called `C`** that represents the temperature that you'd like to convert to Fahrenheit.
- Using the formula provided above, **create a variable called `F`** that is equal to the temperature specified by `C` converted into Fahrenheit.
  - Note that `F` should change automatically if you change what you set `C` equal to.
- **Create two strings** that tell you what the temperature is in Celsius and Fahrenheit. For example:
  - Create one string that is equal to `'Temperature in Celsius: 0 degrees'`, where the `0` is whatever you set `C` equal to.
  - Create one string that is equal to `'Temperature in Fahrenheit: 32 degrees'`, where the `32` is calculated based on what you set `C` equal to.

Data Structures

Data structures are **containers** that hold many values. A few common built-in Python data structures include:

Name	Description	Example
`list`	An **ordered** collection of items. You can put **different data types** in a list. The items have a specific order and are accessed through an **index**. You **can add, delete, or change** items in a list after it is created. Denoted using square brackets, **[ ]**.	`[1, 1, '2', 'hello?', 583,"llama",[2,4,6], 3.14159]`
`tuple`	An **ordered** collection of items. You can put different data types into a tuple. The items have a specific order and are accessed through an **index**. You **cannot add, delete, or change** items in a tuple after it is created. Denoted using parentheses, **`()`**.	`(3,1,4,5,6,7,6,0,7,7)`
`dict`	An **ordered** collection of items that are **accessed via keys**. The data are **stored and accessed via key-value pairs**, like a phonebook: you look up somebody by their name (key) to access their phone number (value). The items have a specific order. You **can add, delete, or change** items in a dictionary after it is created. Denoted using curly braces, **`{}`**, and key:value syntax.	`{'Alice H':13141234567, 'Minerva T':23251352132, 'Tvetene C':3919395021}`

Lists

A list is a collection of items that are **ordered** and **changeable** ("mutable"). Lists are useful because they **can store different data types together**.

Creating a list

A list is created using **square brackets**, [ ]. Items in a list are separated using a comma. An example:

```
my_list = ['apple','banana','cherry','durian']
```

Accessing items in a list

Indexing

You can access a single item in a list using its **index**. An index represents the ***position*** of an item in a collection. Note that in Python, ***indexing starts at 0***. This means that the first item in a list would have the index of 0.

To access an item in a list, take the name of your list followed by the index in square brackets. For example, to access the first item in `my_list` from above, you would write:

```
my_list[0]
```

Which would return:

```
'apple'
```

You can also access items in a list using a **negative index**. A negative index represents the position of an item, but **starting from the back**. This means the last item has an index of -1, second to last -2, etc. For example, you could access the last item in `my_list` using:

```
my_list[-1]
```

Which would return:

```
'durian'
```

A visualization of a list and the different indices for each item:

index	0	1	2	3
`my_list =`	`apple`	`banana`	`cherry`	`durian`
negative index	-4	-3	-2	-1

You can also **index characters within strings**. For example, if `x = 'hello?'`, then `x[-1]` would return: _____

WKSH EX6: Given the following list, answer the questions below and then check your answers in your Jupyter notebook.

```
some_list = [1, 1, '2', 'hello?',
  583, 'llama', [2,4,6], 3.14159]
```

(A) What code would you write to access the 3rd item in the list?

`some_list[2]`

(B) What code would you write to access the last item in the list?

`some_list[-1]`

(C) What code would you write to access the 2nd to last item in the list? What is the data type of this item?

`some_list[-2]`

Slicing

You can also access **multiple items**, or **slices**, of your list by using a **colon (:)** along with the index. The colon specifies the range of the items in your list that you want.

To access a slice in a list, you specify the **starting** and **ending indices** of the items you want, with a **colon in between**. For example, using…

```
my_list = ['apple','banana','cherry','durian']
```

…the following line of code says that we want the items located at index 1 to index 3:

```
my_list[1:3]
```

However, note that when you use the colon to specify a range, this range is ***not inclusive of the number specified on the right side*** - i.e., the `1:3` does not include the value with the index of 3.

index	0	**1**	**2**	3
`my_list =`	`apple`	**`banana`**	**`cherry`**	`durian`

So, `my_list[1:3]` would output the items **`['banana','cherry']`**.

You can also omit either the starting or ending index if you would like to access all items to or from the start or end of the list.

For example, the following line of code…

```
my_list[2:]
```

…would output:

```
['cherry','durian']
```

And the line of code…

```
my_list[:2]
```

…would output:

```
['apple','banana']
```

You can also **slice characters within strings**. For example, if `x = 'hello?'`, then `x[0:2]` would return: **`'he'`**

Changing an item in a list:

To change an item in a list, you first need to access the item in the list via its index, and then set it equal to what you would like to change it to.

For example, to change the first item in `my_list` from above, you would write:

```
my_list[0] = 1234
```

WKSH EX7: Continuing with the list from WKSH EX6, answer the questions below and then check your answers in your Jupyter notebook.

```
some_list = [1, 1, '2', 'hello?',
  583, 'llama', [2,4,6], 3.14159]
```

(A) What code would you write to change the first element of `some_list` to a string representing your name?

```
some_list[0] = alice
```

(B) What code would you write to change the third element of `some_list` to the integer 2?

```
some_list[2] = 2
```

List Methods

Lists are also useful because they come with a range of **methods** that you can use to modify them. A **method** is a built-in function that belongs to a certain kind of object. This means that some methods won't work on all data types.

The syntax for using a method is your variable + a period + the method, with any relevant inputs to that method. In the table below, the following methods are being performed on a list called `my_list`.

Method	Description
`my_list.append()`	Add an item to the end of a list. You'll need to specify the item you are appending to the list.
`my_list.insert()`	Insert an item into a specific position in the list. You'll need to specify the item you are inserting as well as the position you'd like to insert it into.
`my_list.remove()`	Remove an item from a list. You'll need to specify the item you are removing. If there are duplicates in your list, it will only remove the first occurrence of that item.
`my_list.pop()`	Remove an item from a list *by index.* This differs from `remove()` because you specify the *index* of the item you are removing, instead of specifying the item itself.
`my_list.sort()`	Sort the list. If the items in your list are all numbers, it will sort it by ascending numbers. If your items are all strings, it will alphabetize your list. If your list contains a mix of strings and numbers, you may run into an error. You can specify ways to sort. Read documentation for more info.
`my_list.reverse()`	Reverse the order of the items in your list. You don't need to specify anything in this method.

You can also check the length of a list using the built-in function `len()`. This tells you how many items are in the list. Note that you can also use the `len()` function to

`len(some_list)` would return `8`

WKSH EX8: Continuing with the list from WKSH EX6 and 7, answer the questions below and then check your answers in your Jupyter notebook.

```
some_list = [1, 1, '2', 'hello?',
 583, 'llama', [2,4,6], 3.14159]
```

(A) What code would you write if you wanted to add the number `314` after the item `"llama"` in `some_list`?

```
some_list.insert(6,314)
```

(B) How would `some_list` change if you ran the following line of code?

```
some_list.pop(1)
```

It would remove the first instance of 1 in `some_list`.

(C) How would `some_list` change if you ran the following line of code?

```
some_list.append('hi')
```

It would add the string `'hi'` to the end of `some_list`.

Now let's look at some examples of using the methods above in your Jupyter notebook. When you have gone through the examples, continue on to the exercise below.

iPYNB EX6: Say we have a list containing the names of several CSV files:

```
filenames = ['Mod-3-USA-19970101.csv','Mod-3-AUS-19970102.csv',
'Mod-3-BRA-19970103.csv','Mod-3-CAN-19970204.csv','Mod-3-ENG-19970105.csv']
```

(A) You notice that the fourth file, `MetMod-3-USA-19970204.csv`, has the wrong month in the date - it should be `19970104` instead of `19970204`. How would you correct this in `filenames`?

(B) You want to extract the country code of the last file in `filenames`, `'ENG'`. What code could you write to do that?

Tip for (B) - do this in two steps:

1) Extract the file name from `filenames`.
2) Extract the day from the file name.

Tuples

A tuple is a data structure in Python where values are ordered but unchangeable (“immutable”). Like a list, the items are accessed through an index. However, you cannot add, delete, or change items in a tuple after it is created. Tuples are useful if you want to store pieces of information with a specific order that you don’t want to accidentally change - for example, an RGB code or a phone number.

Creating a tuple

A tuple is created using **parentheses**, (). Items in a tuple are separated using a comma. An example:

```
blue = (0,0,1)
```

Accessing items in a tuple

You can access a single item in a tuple using its **index**, with the same syntax as a list. For example:

```
blue[0]
```

would return

```
0
```

Dictionaries

A **dictionary** is a data structure in Python where **values are accessed by keys** instead of indices. Each item in a dictionary is a key:value pair. Dictionaries are useful if you want to store pieces of information that are unique to that key - for example, a person (key) and their phone number (value), a country (key) and an abbreviation you'd like to assign to it (value), or a color (key) and its RGB code (value).

Creating a dictionary

A dictionary is created using **curly braces**, {}, or with the `dict` function. Items in a dictionary are written in key:value pairs and are separated using a comma. An example:

```
colors = {'blue':(0,0,1), 'red':(1,0,0), 'green':(0,1,0)}
```
or
```
colors = dict(blue = (0,0,1), red = (1,0,0), green = (0,1,0)
```

Accessing items in a dictionary

You can access a value in a dictionary using its **key**. To do so, take the name of your dictionary followed by the key in square brackets. For example, to access the value associated with the `blue` entry in `colors`, you would write:

```
colors['blue']
```

Which would return:

```
(0,0,1)
```

Note that every item in a dictionary must have a **unique** key. This means that you **cannot have duplicate keys**. If you use the same key twice, it will take on the last value you set to it.

```
colors = {'blue':(0,0,1), 'blue':(0,0,0.5), 'red':(1,0,0), 'green':(0,1,0)}
```
returns
```
colors = {'blue':(0,0,0.5), 'red':(1,0,0), 'green':(0,1,0)}
```

Why can't you have duplicate keys in a dictionary? **Because Python won't know which data you are associated with the key - e.g., in the example above, it doesn't know whether to return `(0,0,1)` or `(0,0,5)`.**

Adding an item in a dictionary:

To add an item to a dictionary, the syntax is the exact same as changing a value from the dictionary, except you are introducing a **new key** that is not yet in your dictionary and **setting it equal to the new value**. For example:

```
colors['purple'] = (1,1,0)
```
automatically adds the `'purple'` entry into the `colors` dictionary:
```
colors = {'blue':(0,0,0.5), 'red':(1,0,0), 'green':(0,1,0), 'purple':(1,1,0)}
```

iPYNB EX7: Create a dictionary called `constants` that holds the values of the following scientific constants:

pi	R	g
3.141	8.314	9.807

After you have created `constants`, add a key:value pair for `phi`, which is equal to `1.618`.

Control Flow

Booleans

Booleans, or the `bool` data type, can take on only values of **True** or **False**. Booleans are the basis for writing conditional if statements. (You can also use bools for indexing data, but we will not cover this today).

Booleans are the results of **logical operators**. The following are common logical operators:

Less than	Greater than	Less than or equal to	Greater than or equal to	Equal to	Not equal to
`<`	`>`	`<=`	`>=`	`==`	`!=`

Note that the **boolean equal to** "**==**" is different from "=", which is used to assign values to variables (eg, `x = 3`).

You can combine boolean operators to create more specific conditional statements using and/or.	**And**	**Or**
	`&` or `and`	`|` or `or`

For example:

Code	`test = 30 > 10`
	`print(test)`
		`Test = True`
		`Test2 = False`
		`print(Test & Test2)`
			`A = 3`
			`B = 1`
			`print(A>B | A<0)`

Output	`True`	`False`	`True`

WKSH EX9: Evaluate the following by hand. Then, check the answer in your notebook.

(A)

```
x = 2
y = 2
(x+y) < 5
```

`True`

(B)

```
x = 2
y = 2
(x+y < 1)|(x == y)
```

`True`

(C)

```
x = -1
y = 1
(x<0) & (y>0)
```

`True`

Conditional Statements (`if` - `elif` - `else`)

The **if statement** directs Python to **execute certain lines of code (**the executable**)** depending on **whether a certain condition is satisfied**. That **condition** is specified using a **boolean statement**.

The **output of your condition** must always be **`True`** or **`False`** (i.e., a bool). For example:

```
x = 1
if x > 0:
      print('x is greater than 0')
```

If the first condition is not true (i.e., the boolean statement returns `False`), then you may specify any number of additional boolean statements using else ifs (`elif`). Lastly, you have the option to specify an `else`, which will execute if neither the first `if` or any of the following `elif` boolean statements are true.

```
x = 1
if x > 0:
      print('x is positive')
elif x < 0:
      print('x is negative')
else:
      print('x is zero')
```

← Label the conditions and executables in the code.

The syntax for creating an `if` statement is displayed above. Note the following when writing `if` statements:

- The condition in an `if` statement needs to be **ended with a colon**.
- You can use `and` and `or` to **compound multiple conditions**.
- The executable must be **indented to the right** of the `if` statement.
    - If you do not include an indentation, then your conditional won't work; the lines of code in the executable will be executed irrespective of the boolean statement you have specified.

WKSH EX10: Complete the following exercises by hand. Then, check the answer in your Jupyter notebook.

(A)

```
x = <see below>
if x >= 2:
      y = x+2
else:
      y = x-2

print(y)
```

What would **`y`** be if:

`x = 2:` 4

`x = -3.8:` -5.8

(B)

```
T = <see below>
P = <see below>
if (T >= 0) or (P < 10):
      z = 1000
elif (T < 0) and (P > 1):
      z = 500

print(z)
```

What would **`z`** be if:

`T = 10, P = 1:` 1000

`T = -3, P = 0:` 1000

(C) Complete the code below such that `status` equals `'Danger'` if `T` is above 40 or below 0, `'Optimal'` if `T` is between 15 and 30, and `'Acceptable'` in all other cases.

```
T = 20
if (T>40) or (T<0):
      status = 'Danger'
elif (T>=15) and (T<=30):
      status = 'Optimal'
else:
      status = 'Acceptable'

print(instruction)
```

iPYNB EX8: Write a sequence of `if-elif` statements that outputs a student's grade based on their score in the table below. Your code should check the value of `score` and output the variable `grade` based on `score`.

grade	A	B	C	D	F
score	90 ≥ score	80 ≤ score < 90	70 ≤ score < 80	60 ≤ score < 70	score < 60

(A) Define a variable called `score` at the top and set it equal to some score you want to test (e.g., 83). `score` should be used in the condition.
(B) Use a variable called `grade` that is a string (e.g., `'A'`) that depends on the value of `score`.

Hint: You will need to use `and` or `or` to compound multiple conditions.

Loops

The purpose of a **loop** is to **execute the same commands or calculations multiple times** without having to explicitly type it out every single time. Each "loop" through the commands or calculations is called an **iteration**. There are two types of loops: **while** loops and **for** loops. Our exercises will primarily focus on for loops.

For Loops

A **for loop** is used to execute some code for a **predetermined number of iterations**, normally determined by the size of your data set. The for loop is useful when you know how many iterations you need.

A for loop in Python generally has 3 components:

1. The **indexing variable**, which takes on the value of each item in the iterable on each iteration
2. The **iterable**, the collection of things you are iterating through. Determines the number of iterations.
3. The **executable**, the code you are executing on each iteration.

`for j in <iterable>:` `  <executable>`	`j` is called an **indexing variable**. On each iteration, `j` **changes** and **takes on the value of the next value in the iterable** you have specified, and the **executable** is executed. See WKSH EX 11 for several valid ways of setting the iterable.

So, how do we interpret what's going on in a for loop?

`for j in [0,1,2,3]:` `  print(j+1)`	In this example, `[0,1,2,3]` is the iterable. Since there are ___ items in the iterable, this loop will execute the executable ___ times. On each iteration, `j` takes on the value of each item in the iterable. `print(j+1)` is the executable, so this line of code will execute on each iteration.	Complete the table below: Iteration | j | Output 1 2 3 4

The built-in function `range()` is commonly used to create an iterable of sequential numbers from a starting point until stopping point in increments of a step you can specify:

`range(<start>,<stop>,<step>)`

If you don't specify a step, it will default to 1. Note that the iterable created by `range()` is **not inclusive of the right side** - i.e., it creates an iterable from the starting point to the stopping point minus one step. For example:

`range(0,5)` includes `0,1,2,3,4`

`range(0,10,2)` includes `0,2,4,6,8`

WKSH EX11

- How many iterations will each of the following examples go through?
- What value does `j` take on in the **4th iteration** in each example?

(A)	**(B)**	**(C)**

`for j in [0,5,3,7,-2]:`
`  <executable>`
	`for j in range(1,100):`
	`  <executable>`
		`itr = ['a','b','c','d']`
		`for j in itr:`
		`  <executable>`

# total iterations: 5 `j` on 4th iteration: 7	# total iterations: 99 j on 4th iteration: 4	# total iterations: 4 `j` on 4th iteration: `'d'`

WKSH EX12

- How many iterations will each of the following examples go through?
- What value does `j` take on in the **4th iteration** in each example?
- What will the printed output be on the 4th iteration?

(A)

```
x = [10,20,30,40,50]
for j in range(0,5):
      y = x[j]/(j+1)
      print(y)
```

total iterations: 4
`j` on 4th iteration: `40`
Output on 4th iteration: `10`

(B)

```
plants = ['tree','bush','grass','shrub','NA']
for j in range(0,5):
      plant = plants[j]
      print(plant)
```

total iterations: 5
`j` on 4th iteration: `4`
Output on 4th iteration: `'shrub'`

WKSH EX13: The equation for BMI is given by:

$$BMI = \frac{W}{H^2}$$, where H is the height (in cm) and W is the weight (in kg).

Suppose you have the heights and weights of 6 patients stored in the lists `H` and `W`, such that each element in the same position of `H` and `W` corresponds to the same patient (e.g., patient one is 176 cm and weighs 75 kg).

Complete the code below to compute the BMI of each patient, printing the BMI on each iteration.

This pseudocode should help you get started:

- Choose an iterable to represent the **index** (i.e., like in EX12).
- On each iteration, extract the height and weight of patient j (i.e., the j[th] element of both `H` and `W`) and save them to `h_patient` and `w_patient`, respectively.
- Calculate patient j's BMI using the `h_patient` and `w_patient`.
- Print the calculated BMI.

```
H = [1.76,1.62,1.81,1.53,1.70,1.68]
W = [75,65,83,54,91,62]

for j in range(0,6):

      h_patient = H[j]

      w_patient = W[j]



      BMI = h_patient/w_patient**2

      print(BMI)
```

After you have filled out the code, answer the questions below:

total iterations: 6

`j` on 4th iteration: 3

`BMI` on 4th iteration: `23.06805074971165`

If you are having trouble getting started, try filling out the table below.

Iteration	j	h_patient	w_patient	BMI
1				
2				
…	…	…	…	…
n				

While Loops

A **while loop** is used to execute some code for a **conditional number of iterations**, determined by a condition that is checked on each iteration. The while loop is useful when you don't know how many total iterations you need.

When writing a while loop in Python, you set a **condition** that determines how many iterations you are using. Similar to the if statement, the **condition** must return a bool (i.e., `True` or `False`).

`while` **<condition>:** **<executable>**	The executable will be executed as long as the **condition** returns `True`. The **number of iterations** can **depend on both the condition and the executable**, since *the executable may change the condition*. See WKSH EX13 for examples of this.

Note that when writing a loop, your executable must be **indented to the right** of the `while` <condition> line. If you do not include an indentation, then your executable will be executed regardless of your condition.

WKSH EX14

- How many iterations will each of the following examples go through?
- What value does `j` take on in the **4th iteration** in each example?

(A)

```
j = 0
while j < 10:
      j = j+1
```

total iterations: 10
`j` on 4th iteration: `j` starts as `3` and gets changed to `4`

(B)

```
j = 1
while j <= 2:
      j = j-1
```

total iterations: infinite; this loop will go on forever
`j` on 4th iteration: `j` starts as `-2` and gets changed to `-3`

Coding Challenges

Before starting on the exercises below, go to your Jupyter notebook and **look through the sections** on **appending** and **counters** within `for` loops.

Coding Challenge 1: Usernames

Instructions

In this exercise, you have a list of names, `names`, that you need to generate usernames for. Each username must contain the name of the person plus their ID number, stored in `IDs`. For example, the first username would be `'alice142'`. Write a loop that combines creates a username for each item in `names` and `IDs`, and saves each username into the list `usernames`. For example, `usernames` should have a length of 5 and its first element would be `'alice142'`.

Steps

- Use an iterable that represents an index on each iteration (such as in EX12).
  - Hint: recall how to access items in a list using their indices. You may find either the `range()` function or a counter useful for this.
- On each iteration, convert the element in ID into a string.
- On each iteration, concatenate (add together) the string in names and the number in ID to create the username for that person. For example, `'alice' + '142'` would return `'alice142'`.
- On each iteration, append the username you just created to the list `usernames`.
  - You will need the list method `.append()` for this.

```
names = ['alice','clara','alfie','matthew','abdullah','ingo']
ID = [142,753,243,457,503,294]
usernames = []

for _____ in ____________________:
```

If you have no idea where to start, try answering these questions first and filling out the table to the right.

- What is the final output? How many elements should it have at the end?
- How many total iterations do I need to carry out?
  - What different options for iterables can I consider?
- What do I want to accomplish on each iteration? (Is there an equation or computation I need to implement?)

Iteration	Indexing variable	Output
1		
2		
3		
...	...	...
n		

Coding Challenge 2: Experiment Files

Instructions

In this exercise, you have a list of files from different experiments you've conducted, `exp_files`. For each file, we want to extract the date, but only if it is for a file corresponding to experiment A (`expA`). Write a loop that extracts the dates for all the `expA` files only and saves them in a new list called `expA_dates`.

Your final output, `expA_dates`, should return `['20221024','20230814','20210203','20230523']` at the end of your code.

Steps

- Use `exp_files` as the iterable.
- Your code should first check whether the file name has `'expA'` in it using an if statement.
  - Google "check if substring in string Python" and look at some of the Stack Overflow recommendations for going about this.
- On each iteration (that passes the condition above), extract the date from the file name.
  - You might find the string method `.split()` useful.
- On each iteration (that passes the condition above), append the extracted date to `expA_dates`.
  - You will need the list method `.append()` for this.

```
exp_files = ['20221024_expA.xlsx','20231006_expB.csv','20230910_expC.csv',
             '20230814_expA.csv','20220405_expB.xlsx','20211201_expC.xlsx'
            '20210203_expA.xlsx','20220505_expB.csv','20211102_expC.xlsx'
            '20230523_expA.xlsx','20230611_expB.csv','20211102_expC.xlsx']


expA_dates = []

for _____ in ____________________:
```

If you have no idea where to start, try answering these questions first and filling out the table to the right.

- What is the final output? How many elements should it have at the end?
- How many total iterations do I need to carry out?
  - What different options for iterables can I consider?
- What do I want to accomplish on each iteration? (Is there an equation or computation I need to implement?)

Iteration	Indexing variable	Output
1		
2		
3		
...	...	...
n		

Coding Challenge 3: Temperature Converter

Instructions

In this exercise, you will write some code that converts a series of temperature measurements into Celsius. You have a series of temperature measurements stored in a list called `T`. The corresponding unit of each of the measurements is stored in a separate list called `units`. Your goal is to convert all of the measurements in `T` into Celsius, and store them to a list called `T_C`. Your final output, `T_C`, should be a list containing all of the converted temperatures, with the same number of elements as `T` and `units`.

The formulae for converting from Fahrenheit and Kelvin are:

$$C = \frac{5}{9} \cdot (F - 32)$$

$$C = K - 273.15$$

Steps

- Use an iterable in your `for` loop that represents an index on each iteration (such as in EX12).
  - *Hint: recall how to access items in a list using their indices.*
  - You may find either the `range()` function or a counter useful for this.
- On each iteration, you should have a series of `if` statements that check the unit of that measurement in `units`.
  - Based on the unit, apply the correct temperature conversion formula.
- After applying the correct temperature conversion, save the converted value into `T_C`.

```
T = [30,280,14,85,58,317,251,305,5,69,301]
units = ['C','K','C','F','F','K','K','K','C','F','K']

T_C = []


for _____ in ____________________:
```

If you have no idea where to start, try answering these questions first and filling out the table to the right.

- What is the final output? How many elements should it have at the end?
- How many total iterations do I need to carry out?
  - What different options for iterables can I consider?
- What do I want to accomplish on each iteration? (Is there an equation or computation I need to implement?)

Iteration	Indexing variable	Output
1		
2		
3		
...	...	...
n		

Functions

Functions generally take inputs (called **parameters**) and return one or more outputs. Functions are useful because they are reusable - you can run the same code on different variables without having to rewrite all the code every time.

The syntax for creating a function in Python is as follows:

```
def fcn(input_1, input_2,...input_n):
      output_1 = some operation using any combination of input_1,input_2,...input_n
      output_2 = some operation using any combination of input_1,input_2,...input_n
      ...
      output_m = some operation using any combination of input_1,input_2,...input_n
      return output_1, output_2,...output_m
```

For example, simple mathematical functions can be represented by a Python function like so:

`def f(x):`
`  y = x+1`
`  return y`
	`def g(x,y):`
	`  z = x**2+**2`
	`  return z`

What would `f(4)` return? _____	What would `g(1,0)` return? _____

Note the following when writing a function:

- You must use the `def` keyword, followed by your **function name**, the **inputs** you would like to use in **parentheses**, and followed by a **colon**.
- The operations that you perform within your function must be **indented to the right** of the `def` line with your function name and inputs.
- You must specify the **outputs** you would like your function to compute using the `return` keyword.

WKSH EX15

```
def add_max(list1,list2):

      list1_max = max(list1)
      list2_max = max(list2)

      list1_2_max = list1_max + list2_max

      return list1_2_max
```

(A) What is this function called? `add_max`

(B) What inputs does this function take?
`list1, list2`

(C) What operations does this function perform on its inputs?
This function finds the maxima of both the input lists and then adds them together.

(D) What output(s) does this function produce?
`list1_2_max`

(E) What output would you get if you ran the code `add_max([1,0.5,-1,3],[10,13])`?
16

Another key aspect about writing functions is that the variables you create within a function **do not exist in your workspace** - they are only ***defined locally***.

For example, in the **`add_max`** function above, the variables `list1_max` and `list2_max` were created to compute the output. Say we ran the piece of code in **1e**:

```
add_max([1,0.5,-1,3],[10,13])
```

You might think that this means there are the variables `list1_max` (equal to 3) and `list_2max` (equal to 13) in your workspace. However, if you tried to run code with these variables, it would tell you that they don't exist. That's because these variables are *only defined locally* - i.e., within the context of your function.

While you might find this feature of functions difficult to work with right now, this can actually save memory when doing heavy computations.

WKSH EX16: Complete the following exercises in the worksheet. Then, type them into your Jupyter notebook.

(A) Write a function called `parabola` that takes an input `x` and outputs a point on the parabola $y = x^2+2x+1$.

```
def parabola(x):
    y = x**2+2*x+1

    return y
```

(B) Write a function called `circles` that takes a radius of a circle as input and returns a) the circumference, and b) the area of that circle.

```
def circles(r):
    circ = 2*3.14159*r
    area = 3.14159*(r**2)

    return circ,area
```

iPYNB EX9: In **iPYNB EX8**, you wrote a sequence of `if-elif` statements that outputs a student's grade based on their score in the table below.

grade	A	B	C	D	F
score	90 ≥ score	80 ≤ score < 90	70 ≤ score < 80	60 ≤ score < 70	score < 60

Using your code from this exercise, create a function called `my_grade()` that takes `score` as an input and outputs the grade.