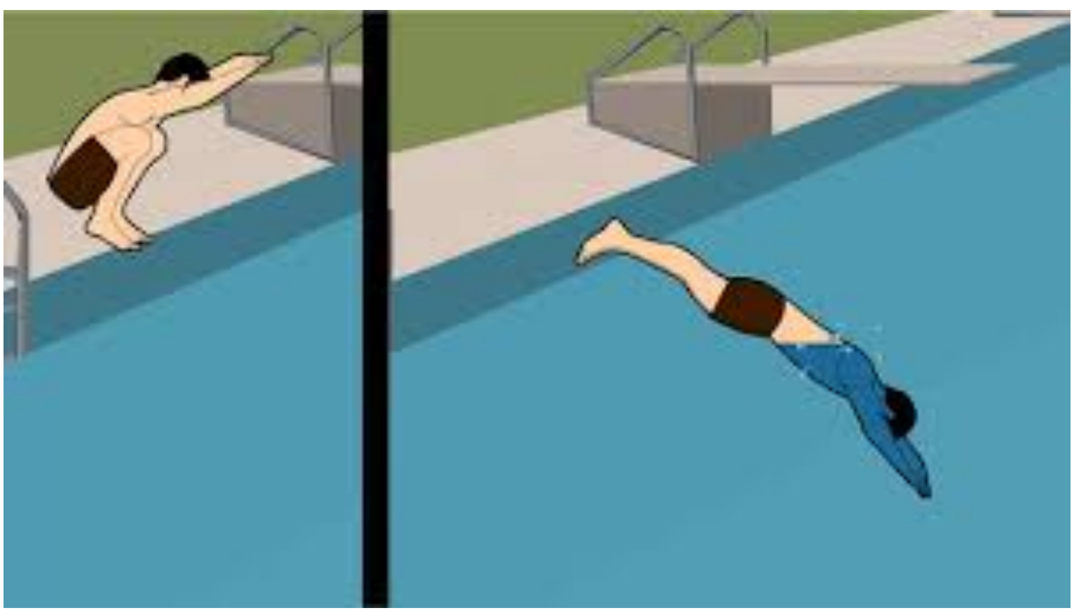
In [1]:
```
%%html
<link rel="stylesheet" type="text/css" href="rise.css" />
```

Python basics

How to learn python

1. Learn the very very basics.
2. **Dive in and apply it**, learning more advanced features as you go.



3. **Practice, practice, practice.** The homework is your friend!

Basic data types

Learning goals

- You will understand python's basic data types.

In [2]:
```
type(82)
```
Out[2]:
```
int
```

In [3]:
```
type(3.14)
```
Out[3]:
```
float
```

In [4]:
```
type(True), type(False)
```
Out[4]:
```
(bool, bool)
```

In [5]:
```
type("Hello"), type('hi'), type("y'all")
```
Out[5]:
```
(str, str, str)
```

Variables

Learning goals

- You will be able to assign values to variables.
- You will appreciated the importance of variable names.

Variable names can include **letters**, **numbers** and **underscores**, but *cannot start with a number*.

In [6]:
```
x = 5
value9 = -34.2
my_str = "hello"
_82Flag = True
```

In [7]:
```
x, value9, my_str, _82Flag
```
Out[7]:
```
(5, -34.2, 'hello', True)
```

Operations with variables are the same as with values.

In [8]:
```
value9 >= x
```
Out[8]:
```
False
```

In [9]:
```
x * 5
```
Out[9]:
```
25
```

Variable names are important!

A huge amount of writing code **comes down to naming things!**

Good variable names are an important part of making your code understandable.

Which of the following variable names would you prefer?

In [10]:
```
a = 54.6
theSpikingRateOfFosExpressingNeuronsInVentralCA1 = 54.6
spike_rate_fos = 54.6
```

Variable names

- **too short**: *unclear unless only used in immediate vicinity*
- **too long**: *makes it hard to read super long code lines*
- **just right**:

Special variable names

`__builtins__`, `__dict__`, `__init__`, ...

Variable names *surrounded by two consecutive underscores* on each side are used for special Python variables.

You should probably NOT use this naming pattern for your own variables as you will both confuse people and potentially overwrite a Python variable.

Unless you know what you are about, I would avoid using underscores for anything other than separating consecutive words in a variable name.

Format strings

Learning goals

- You will be able to insert formatted values into strings.

In [11]:
```
x = -56.359218
y = 2

f"A string that includes the values {x} and {y}."
```

Out[11]:
```
'A string that includes the values -56.359218 and 2.'
```

Formatting options

In [12]:
```
x = -56.359218

f"A string that includes the floating point value {x:.2f} with two decimal p
```

Out[12]:
```
'A string that includes the floating point value -56.36 with two decimal pl
aces.'
```

Comments

Learning goals

- You will from now on always comment your code so everyone can understand it *(including yourself later on)* ;)

In [13]:
```
# number of neurons in dataset
n = 156  # so many neurons!
```

In [14]:
```
""" A shorthand variable like `n` can provide very readable code
if it is only used in a local context (e.g. this cell).

However, if you refer to `n` in later cells,
it may not be obvious what you mean by it.
"""

# these names are much more clear if used later
num_neurons = 156  # snake_case
numNeurons = 156  # camelCase
NumNeurons = 156  # PascalCase
```

In [15]:
```
"""
Although super clear, writing any sort of expression
involving this variable will end up in a really long line of code
that can make the code difficult to read.
"""
theNumberOfThoseRareNeuronsWithHighlyDistinctProcesses = 156
```

Multi-line strings

In [16]:
```
mystr = """A multi-line
string stored

in a variable."""

print(mystr)
```

```
A multi-line
string stored

in a variable.
```

Understandable code

Variable Names + Comments = *Understandable*

You will spend **much more time reading code** (*including your own*) than writing it.

Well commented code with **informative variable names** is essential for others (*and also yourself a month or a year from now*) to understand what your code does.

Basic operations

Learning goals

- You will be able to do math and concatenate strings.

In [17]:
```python
print(3 + 2)
print(3 - 2)
print(3 * 2)
print(3 / 2)
print(3**2)  # !!! NOT 3^2
```

```
5
1
6
1.5
9
```

Whitespace does NOT matter!

In [18]:
```python
print(3 + 2)
print(3-2)
print( 3 * 2 )
print(3    /2)
print (3 ** 2)
```

```
5
1
6
1.5
9
```

Whitespace does NOT matter (except for *new lines*).

In [19]:
```
print(3 + 2)
print(3 - 2)
```

```
5
1
```

In [20]:
```
print(3 + 2) print(3 - 2)
```

```
  Cell In[20], line 1
    print(3 + 2) print(3 - 2)
                 ^
SyntaxError: invalid syntax
```

Whitespace does NOT matter (except for *new lines* and *line indentation*).

In [21]:
```
print(3 + 2)
print(3 - 2)
```

```
5
1
```

In [22]:
```
print(3 + 2)
    print(3 - 2)
```

```
  Cell In[22], line 2
    print(3 - 2)
    ^
IndentationError: unexpected indent
```

A few more operations.

In [23]:
```
(3 + 2) * 2
```

Out[23]: 10

In [24]:
```
"howdy" + " there"
```

Out[24]: 'howdy there'

In [25]:
```
# floor division

11 // 4
```

Out[25]: 2

In [26]:
```
# modulus = remainder after floor division

11 % 4
```

Out[26]: 3

Operations should make sense.

In [27]:
```
5 + 2
```

Out[27]: 7

In [28]:
```
5 + "two"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[28], line 1
----> 1 5 + "two"

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In [29]:
```
5 * "two"
```

Out[29]: 'twotwotwotwotwo'

Some shorthand for operations on variables.

In [30]:
```
x = 3
```

In [31]:
```
# x = x + 2
x += 2
x
```

Out[31]: 5

In [32]:
```
x -= 2
x
```

Out[32]: 3

In [33]:
```
x *= 2
x
```

Out[33]: 6

In [34]:
```
x /= 2
x
```

Out[34]: 3.0

Logical comparison

Learning goals

- You will be able to compare things.

In [35]:
```
3 == 3
```

Out[35]: True

In [36]:
```
3 != 3
```

Out[36]: False

In [37]:
```
3 > 4
```

Out[37]: False

In [38]:
```
3 <= 3
```

Out[38]: True

In [39]:
```
"hi" == "hello", "hi" != "hello"
```

Out[39]: (False, True)

In [40]:
```
3 < 4 and 3 == 4
```

Out[40]: False

In [41]:
```
3 == 4 or 3 <= 4
```

Out[41]: True

In [42]:
```
not 3 == 4
```

Out[42]: True

In [43]:
```
3 < 4 and not (3 == 4 or 4 <= 3)
```

Out[43]: True

This is pretty neat!

In [44]:
```
x = 3

2 < x < 4
```

Out[44]:
```
True
```

Exercise

On average, neuron A spikes 3.4 times per second, whereas neuron B spikes 11.9 times per second.

1. Assign two variables for the average spike rates of neurons A and B, respectively.
2. Use your two variables to assign a third variable for whether or not neuron A spikes faster that neuron B on average.

In [ ]:
```

```

Exercise Key

On average, neuron A spikes 3.4 times per second, whereas neuron B spikes 11.9 times per second.

1. Assign two variables for the average spike rates of neurons A and B, respectively.
2. Use your two variables to assign a third variable for whether or not neuron A spikes faster that neuron B on average.

In [45]:
```
# These variable names are just an example.
# It is OK if your variable names differ,
# so long as they are understandable.

spike_rate_A = 3.4
spike_rate_B = 11.9
is_A_faster_than_B = spike_rate_A > spike_rate_B

is_A_faster_than_B
```

Out[45]:
```
False
```

Conditional code blocks

Learning goals

- You will understand python's code block architecture.
- You will be able to conditionally execute code blocks.

In [46]:
```python
execute_block = False

if execute_block:
    print("I'm in the block.")
    print("I'm also in the block.")

    print("I'm still in the block.")
print("I'm NOT in the block.")
```

```
I'm NOT in the block.
```

In [47]:
```python
do_this = True

if do_this:
    print("I'm in the block.")
    print("I'm also in the block.")

    print("I'm still in the block.")
print("I'm NOT in the block.")
```

```
I'm in the block.
I'm also in the block.
I'm still in the block.
I'm NOT in the block.
```

`if-elif-else` code block

In [48]:
```python
animal = "fish"

if animal == "cat":
    print('meow')
elif animal == "dog":

    print('bark')

elif animal == "bird":
    pass  # do nothing
else:
    print('not a cat')
```

```
    print('not a dog')
    print('not a bird')
```

```
not a cat
not a dog
not a bird
```

Exercise

Write code that prints "negative", "zero" or "positive" depending on whatever value x is assigned.

In [49]:
```
x = -2.1

# your code here...
```

Exercise Key

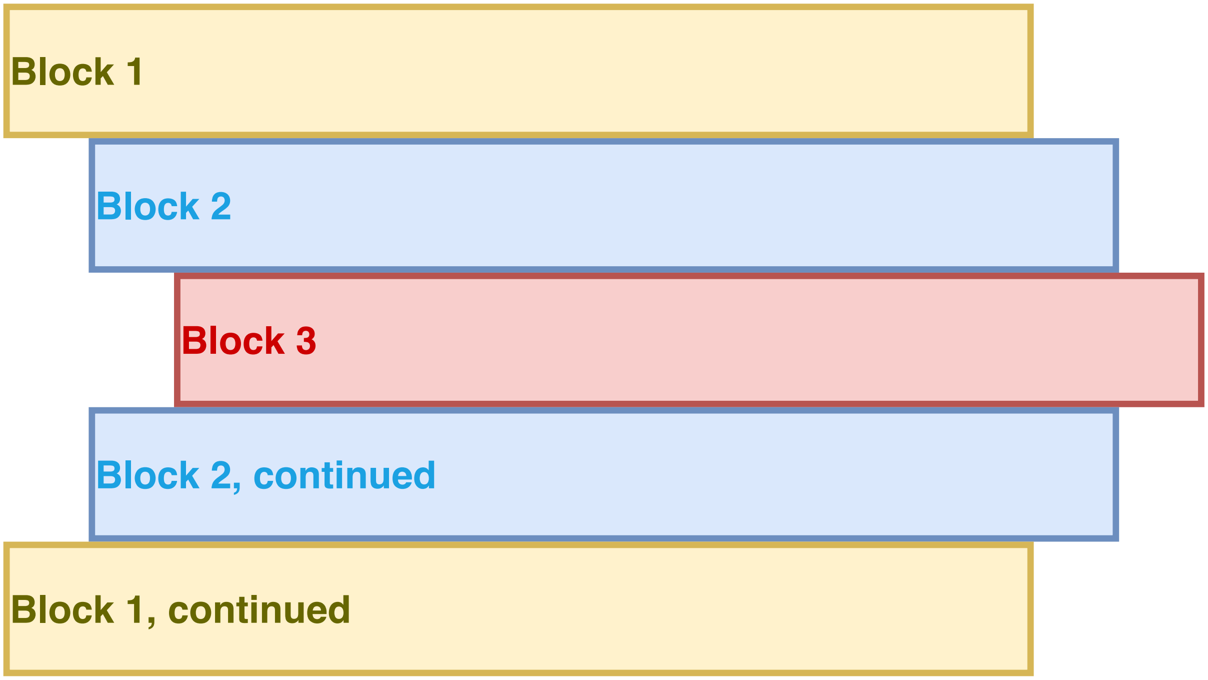
Write code that prints "negative", "zero" or "positive" depending on whatever value x is assigned.

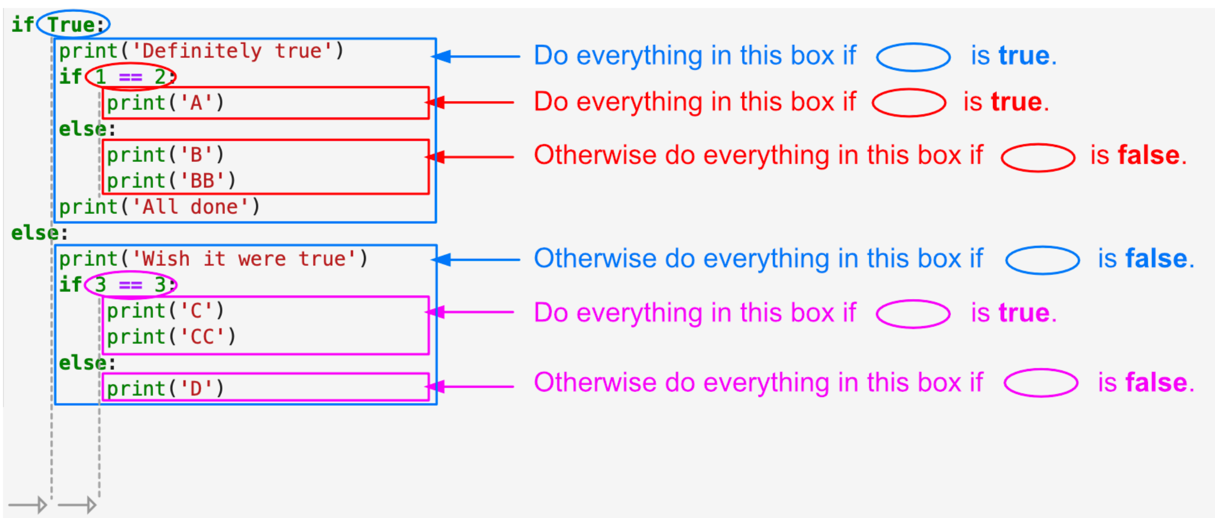
In [50]:
```
x = -2.1

if x == 0:
    print("zero")
elif x < 0:
    print("negative")
else:
    print("positive")
```

```
negative
```

Nested code blocks

Indentation defines how **Python organizes all code into blocks**.





List

Learning goals

- You will be able to work with lists of items.
- You will be able to index/slice into a list.

In [2]:
```
alist = [1, 2, 3.4, "hi", True]
alist
```

Out[2]:
```
[1, 2, 3.4, 'hi', True]
```

In [3]:
```
len(alist)
```

Out[3]: 5

In [4]:
```
blist = []
len(blist)
```

Out[4]: 0

In [5]:
```
blist.append(2)           # append a single value
blist.extend([5.4, False])  # extend by a list of values
blist
```

Out[5]: [2, 5.4, False]

In [6]:
```
blist.remove(2)
blist
```

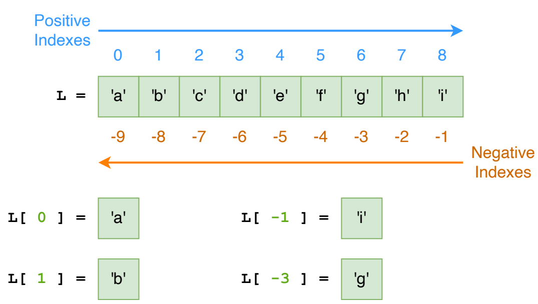
Out[6]: [5.4, False]

List unpacking

In [7]:
```
mylist = [1, "hi", True]

a, b, c = mylist

print(a)
print(b)
print(c)
```

```
1
hi
True
```

List indexing



In [8]:
```
mylist = [1, 2, 3.4, "hi", True]
```

```
mylist[0]
```

Out[8]: 1

In [9]:
```
mylist[0], mylist[1], mylist[-1], mylist[-2]
```

Out[9]: (1, 2, True, 'hi')

Exercise

Use list indexing to assign the specified subjects in from the given list of subjects.

In [10]:
```
subjects = ["M0159", "F0287", "FBEST", "MALZ", "FOG17"]

# first_subject = ...
# second_subject = ...
# second_to_last_subject = ...
# last_subject = ...
```

Exercise Key

Use list indexing to assign the specified subjects in from the given list of subjects.

In [11]:
```
subjects = ["M0159", "F0287", "FBEST", "MALZ", "FOG17"]

first_subject = subjects[0]
second_subject = subjects[1]
second_to_last_subject = subjects[-2]
last_subject = subjects[-1]
```

You can use a variable to index into a list.

In [12]:
```
mylist = [1, 2, 3.4, "hi", True]

i = 3

mylist[i]
```

Out[12]: 'hi'

You can edit list items via their index.

In [13]:
```python
mylist[i] = "bye"
mylist[0] = mylist[0] - 2

mylist
```

Out[13]: `[-1, 2, 3.4, 'bye', True]`

You can insert and delete list items via their index.

In [14]:
```python
mylist = [1, 2, 3.4, "hi", True]

mylist.insert(1, 800)

mylist
```

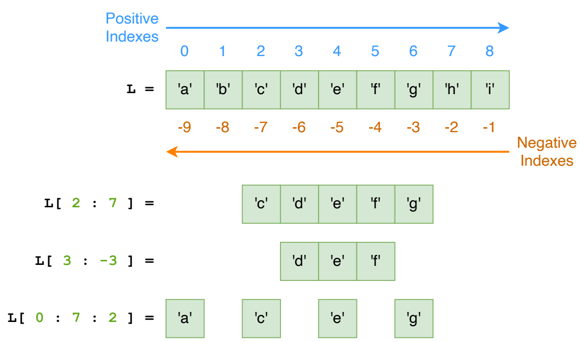
Out[14]: `[1, 800, 2, 3.4, 'hi', True]`

In [15]:
```python
del mylist[-3]

mylist
```

Out[15]: `[1, 800, 2, 'hi', True]`

List slicing

```
list[start:stop]
list[start:stop:step]
```




```
list[start:stop]
list[start:stop:step]
```

In [16]:
```python
mylist = [1, 2, 3.4, "hi", True, 5, 82, 99]

# 1:4 ==> 1,2,3
```

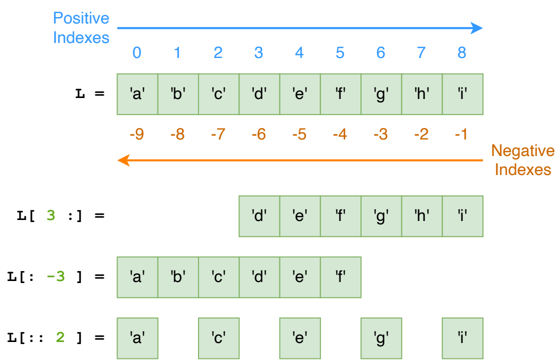
```
mylist[1:4]
```

Out[16]: [2, 3.4, 'hi']

In [17]:
```
# 1:7:2 ==> 1,3,5

mylist[1:7:2]
```

Out[17]: [2, 'hi', 5]

List slice defaults



In [18]:
```
mylist = [1, 2, 3.4, "hi", True, 5, 82, 99]

mylist[4:]  # index 4 and everything after
```

Out[18]: [True, 5, 82, 99]

In [19]:
```
mylist[:4]  # everything before index 4
```

Out[19]: [1, 2, 3.4, 'hi']

In [20]:
```
mylist[:]  # everything
```

Out[20]: [1, 2, 3.4, 'hi', True, 5, 82, 99]

In [21]:
```
mylist[::2]  # every other item
```

Out[21]: [1, 3.4, True, 82]

Exercise

Use list slicing to get the odd or non-negative even numbers from the specified list.

In [22]:
```
numbers = [-2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# odds = numbers[...]
# non_neg_evens = numbers[...]

# odds, non_neg_evens
```

Exercise Key

Use list slicing to get the odd or non-negative even numbers from the specified list.

In [23]:
```
numbers = [-2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

odds = numbers[1::2]
non_neg_evens = numbers[2::2]

odds, non_neg_evens
```

Out[23]:
```
([-1, 1, 3, 5, 7, 9], [0, 2, 4, 6, 8])
```

Nested list

In [24]:
```
mylist = [1, 2, [3, 4]]

mylist
```

Out[24]:
```
[1, 2, [3, 4]]
```

In [25]:
```
mylist[2]
```

Out[25]:
```
[3, 4]
```

In [26]:
```
mylist[2][-1]
```

Out[26]:
```
4
```

!!! This quickly becomes cumbersome with more than a few levels of nesting.

List comprehension

Learning goals

- You will be able to use simple list comprehensions.

`[expression for item in collection]`

In [27]:
```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

numbers_squared = [x**2 for x in numbers]

numbers_squared
```

Out[27]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

`[expression for item in collection if condition]`

In [28]:
```
[y * 3 for y in numbers if y < 5]
```

Out[28]:
```
[0, 3, 6, 9, 12]
```

!!! I recommend avoiding these except in the simplest cases as they can become hard to read.

Exercise

Use a list comprehension to get all numbers in the range 2 to 7 from the specified list.

In [29]:
```
numbers = [-2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# num2to7 = [...]

# num2to7
```

Exercise Key

Use a list comprehension to get all numbers in the range 2 to 7 from the specified list.

In [30]:
```
numbers = [-2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

num2to7 = [x for x in numbers if x >= 2 and x <= 7]

num2to7
```

Out[30]:
```
[2, 3, 4, 5, 6, 7]
```

Dictionary

Learning goals

- ou will be able to work with a dictionary of (key, value) pairs.
- You will be able to values by their key.

In [31]:
```
params = {
    "cell-type": "neuron",
    "region": "CA1",
    "Fos-expression": 0.62,
    "number of cells": 89
    }

params
```

Out[31]:
```
{'cell-type': 'neuron',
 'region': 'CA1',
 'Fos-expression': 0.62,
 'number of cells': 89}
```

Values are indexed via their keys.

In [32]:
```
params["region"]
```

Out[32]:
```
'CA1'
```

Keys can be anything.

In [33]:
```
params[100.5] = [87, True]

params
```

Out[33]:
```
{'cell-type': 'neuron',
 'region': 'CA1',
 'Fos-expression': 0.62,
 'number of cells': 89,
 100.5: [87, True]}
```

Values can be edited/deleted via their keys.

In [34]:
```
params
```

Out[34]:
```
{'cell-type': 'neuron',
 'region': 'CA1',
 'Fos-expression': 0.62,
 'number of cells': 89,
 100.5: [87, True]}
```

In [35]:
```
params["region"] = "cortex"

del params[100.5]

params
```

Out[35]:
```
{'cell-type': 'neuron',
 'region': 'cortex',
 'Fos-expression': 0.62,
 'number of cells': 89}
```

Exercise

Edit the specified parameters dictionary as indicated.

In [36]:
```
params = {
    "cell-type": "neuron",
    "region": "CA1",
    "Fos-expression": 0.62,
    "number of cells": 89
    }

# 1. Change the cell type to 'interneurons'.

# 2. Add a new parameter for the number of subjects used in the study (e.g.,
```

Exercise Key

Edit the specified parameters dictionary as indicated.

In [37]:
```
params = {
    "cell-type": "neuron",
    "region": "CA1",
    "Fos-expression": 0.62,
    "number of cells": 89
    }

# 1. Change the cell type to 'interneurons'.
params["cell-type"] = "interneurons"

# 2. Add a new parameter for the number of subjects used in the study (e.g.,
params["num_subjects"] = 12

params
```

Out[37]:
```
{'cell-type': 'interneurons',
 'region': 'CA1',
 'Fos-expression': 0.62,
 'number of cells': 89,
 'num_subjects': 12}
```

Iterating lists and dictionaries

Learning goals

- You will be able to iterate over the items in a list or dictionary using a `for` loop.
- You will be able to iterate by value or by index.
- You will be introduced to the `len()` and `range()` functions.
- You will be able to `break` out of the loop or skip to `continue` to the next item.
- You will be able to loop over items `while` a condition remains true.

`for` loop code block

In [38]:
```
mylist = [1, 2, 3.4, "hi", [5, 6], True]

for item in mylist:
    print(item)
```

```
1
2
3.4
hi
[5, 6]
True
```

You can name the loop variable whatever you want.

In [39]:
```python
for x in mylist:
    print(x)
```

```
1
2
3.4
hi
[5, 6]
True
```

The entire code block is executed for each item.

In [40]:
```python
mylist = [1, 2, 3.4, "hi", [5, 6], True]

for item in mylist:
    if item == "hi":
        item = "bye"
    else:
        item = item * 3
    print(item)
```

```
3
6
10.2
bye
[5, 6, 5, 6, 5, 6]
3
```

range function

In [41]:
```python
for i in range(4):
    print(i)
```

```
0
1
2
3
```

In [42]:
```python
for i in range(1, 4):
    print(i)
```

```
1
2
3
```

In [43]:
```
for i in range(1, 4, 2):
    print(i)
```

```
1
3
```

help function

In [44]:
```
help(range)
```

```
Help on class range in module builtins:

class range(object)
 |  range(stop) -> range object
 |  range(start, stop[, step]) -> range object
 |
 |  Return an object that produces a sequence of integers from start (inclus
ive)
 |  to stop (exclusive) by step.  range(i, j) produces i, i+1, i+2, ..., j-
1.
 |  start defaults to 0, and stop is omitted!  range(4) produces 0, 1, 2, 3.
 |  These are exactly the valid indices for a list of 4 elements.
 |  When step is given, it specifies the increment (or decrement).
 |
 |  Methods defined here:
 |
 |  __bool__(self, /)
 |      True if self else False
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __gt__(self, value, /)
 |      Return self>value.
```

```
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __reduce__(...)
 |      Helper for pickle.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __reversed__(...)
 |      Return a reverse iterator.
 |
 |  count(...)
 |      rangeobject.count(value) -> integer -- return number of occurrences
of value
 |
 |  index(...)
 |      rangeobject.index(value) -> integer -- return index of value.
 |      Raise ValueError if the value is not present.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signatu
re.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  start
 |
 |  step
 |
```

```
|  stop
```

Iterate over indexes.

In [45]:
```python
mylist = [1, 2, 3.4, "hi", [5, 6], True]

N = len(mylist)

for i in range(N):
    print(i, mylist[i])
```

```
0 1
1 2
2 3.4
3 hi
4 [5, 6]
5 True
```

In [46]:
```python
for i, val in enumerate(mylist):
    print(i, val)
```

```
0 1
1 2
2 3.4
3 hi
4 [5, 6]
5 True
```

`break` and `continue`

In [47]:
```python
for i in range(10):
    if i == 3:
        continue  # skip to next loop iteration
    if i == 7:
        break  # exit the loop
    print(i)
```

```
0
1
2
4
5
6
```

`while` loop code block

In [48]:
```python
i = 0
```

```python
while i < 5:
    print(i)
    i += 1
```

```
0
1
2
3
4
```

Stuck in an infinite loop? *Stop the kernel.*

In [49]:
```python
i = 0
while i < 5:
    j = i**2
    # oops, forgot to increment i!
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[49], line 2
      1 i = 0
----> 2 while i < 5:
      3     j = i**2
      4     # oops, forgot to increment i!

KeyboardInterrupt: 
```

Iterating over the keys in a dictionary.

In [50]:
```python
params = {
    "cell-type": "neuron",
    "region": "CA1",
    "Fos-expression": 0.62,
    "number of cells": 89
    }

for key in params:
    print(key, "=", params[key])
```

```
cell-type = neuron
region = CA1
Fos-expression = 0.62
number of cells = 89
```

Assignment vs. mutation

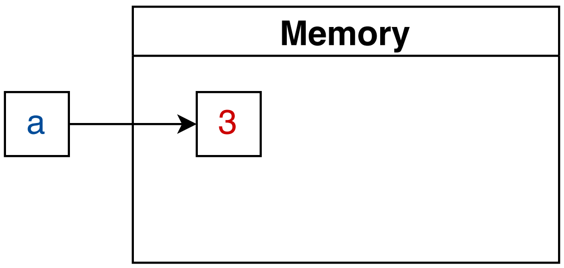
Learning goals

- You will understand what happens when you assign a value to a variable.
- You will understand the difference between mutating a value in memory and assigning a new value.

Assigning a value to a variable creates a new value in memory.

In [51]:
```
a = 3

id(a)
```

Out[51]: 4313639024

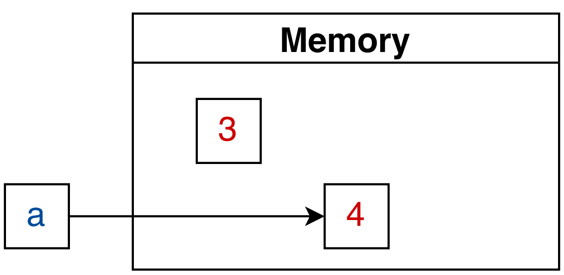


Reassigning a new value to a previously defined variable creates a new value in memory.

!!! It does NOT have to overwrite the memory for the previous value that the variable referred to.

In [52]:
```
a = 4

id(a)
```

Out[52]: 4313639056

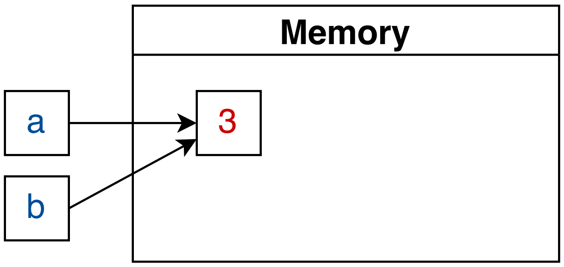


Multiple variables can refer to the same value in memory.

In [53]:
```
a = 3
```

```python
b = a

id(a) == id(b)
```

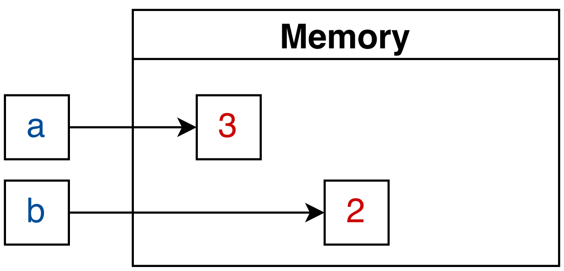
Out[53]: True



!!! Reassigning a variable has NO EFFECT on other variables even if they previously referred to the same memory location.

In [54]:
```python
b = 2

a, id(a) ==  id(b)
```

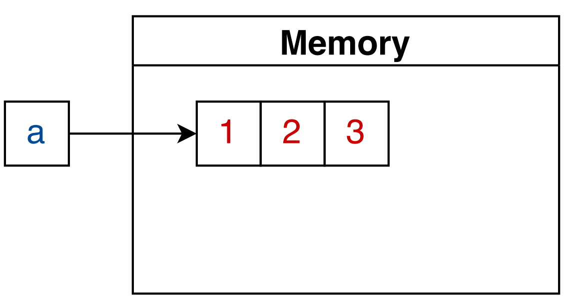
Out[54]: (3, False)



Assigning a value to a variable creates a new value in memory.

In [55]:
```python
a = [1, 2, 3]

id(a)
```

Out[55]: 4372598784

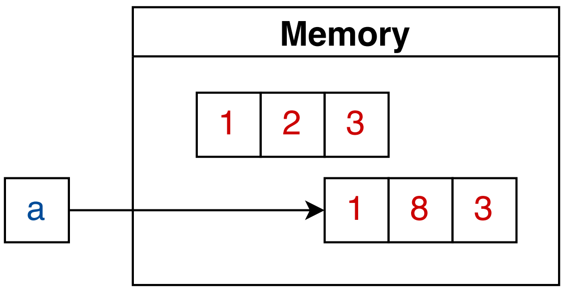


Reassigning a new value to a previously defined variable creates a new value in memory.

!!! It does NOT have to overwrite the memory for the previous value that the variable referred to.

In [56]:
```
a = [1, 8, 3]

id(a)
```

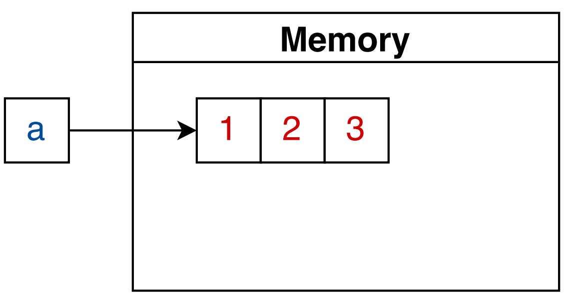
Out[56]: 4372615232



Let's start by assigning a simple list.

In [57]:
```
a = [1, 2, 3]

id(a)
```

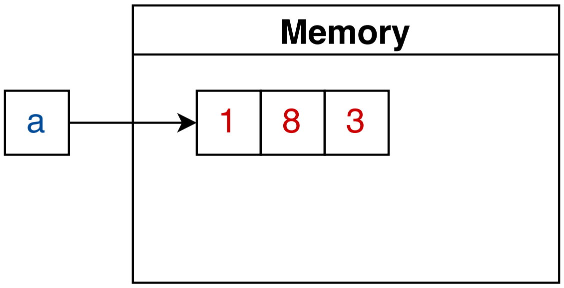
Out[57]: 4367511424



!!! Mutating a variable changes the value in the current memory location.

In [58]:
```
a[1] = 8

id(a)
```

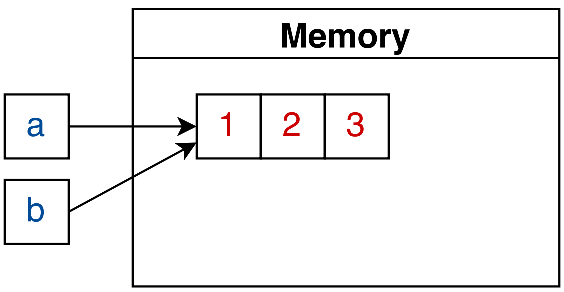
Out[58]: 4367511424



Multiple variables can refer to the same value in memory.

In [59]:
```
a = [1, 2, 3]
b = a

id(a) == id(b)
```

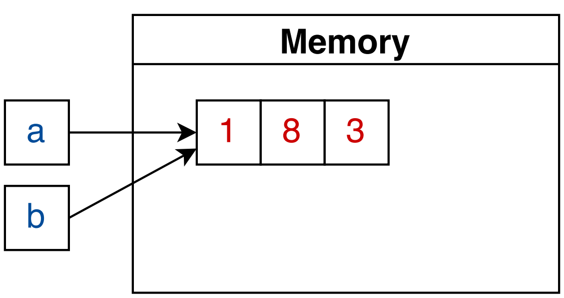
Out[59]: True



!!! Mutating a variable changes ALL VARIABLES that refer to the same value in memory.

In [60]:
```
a[1] = 8

b, id(a) == id(b)
```

Out[60]: ([1, 8, 3], True)



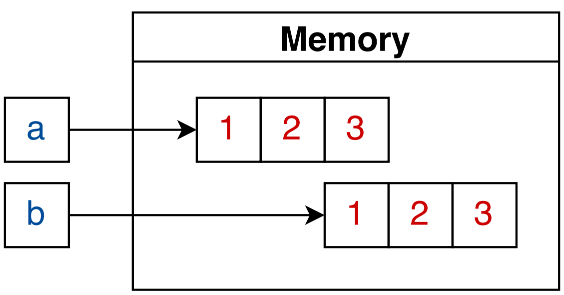
Examples of **mutable** vs. **immutable** objects.

Immutable	Mutable
Can only be assigned.	Can be assigned and mutated.
`int`	`list`
`float`	`dict`
`bool`	`class`
`str`	
`tuple`	

What if you want a **copy** of a mutable object?

In [61]:
```
# import the copy module
import copy

a = [1, 2, 3]

# use the 'copy' funtion in the 'copy' module
b = copy.copy(a)

b, id(a) == id(b)
```

Out[61]:
```
([1, 2, 3], False)
```



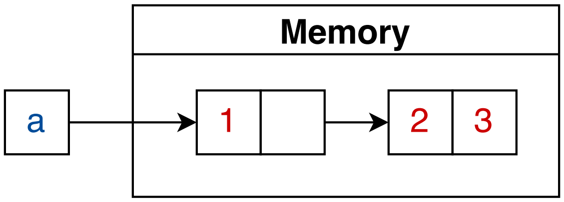
`copy.deepcopy` is needed for recursive copying of nested objects.

In [62]:
```
a = [[1, 2], [3, 4]]

b = copy.deepcopy(a)

b, id(a) == id(b)
```

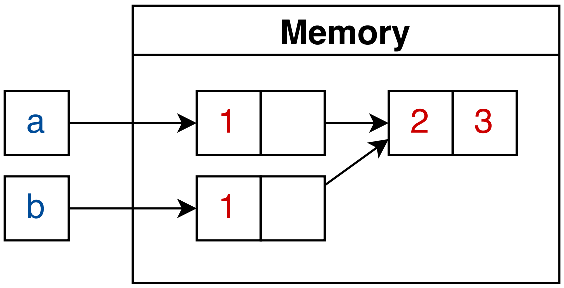
Out[62]: ([[1, 2], [3, 4]], False)

Nested objects in memory.

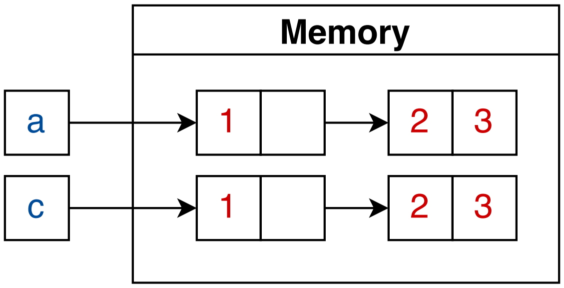
In [63]:
```
a = [1, [2, 3]]
```



In [64]:
```
import copy

b = copy.copy(a)
```



In [65]:
```
c = copy.deepcopy(a)
```



Tuple = Immutable List

In [66]:
```
mylist = [1, 2, 'hi', True]
mytuple = (1, 2, 'hi', True)

type(mylist), type(mytuple)
```

Out[66]: (list, tuple)

You can read the values in a tuple using the same indexing/slicing as for a list.

In [67]:
```
mylist[1]
```

Out[67]: 2

In [68]:
```
mytuple[2]
```

Out[68]: 'hi'

Unlike a list, you CANNOT CHANGE a tuple.

In [69]:
```
mylist[1] = 100
mylist
```

Out[69]: [1, 100, 'hi', True]

In [70]:
```
mytuple[1] = 100
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[70], line 1
----> 1 mytuple[1] = 100

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

You can ONLY ASSIGN A COMPLETELY NEW tuple.

In [71]:
```
mytuple = (1, 100, 'hi', True)
```

Functions

Learning goals

- You will be able to write and use your own functions.
- You will be able to define named and/or default arguments.
- You will be able to `return` an output from a function.
- You will understand a variable's scope.

In [72]:
```
def sayHi(name):
    message = "Hi " + name
    print(message)

sayHi("Tim")
```

```
Hi Tim
```

In [73]:
```
sayHi("Amy")
```

```
Hi Amy
```

In [74]:
```
type(sayHi)
```

Out[74]:
```
function
```

The function name is always followed by `()` even if the function does not take any input arguments.

In [75]:
```
def sayHi():
    print("Hi")

sayHi()
```

```
Hi
```

Function return values

In [76]:
```
def sum(x, y):
    return x + y

z = sum(3, 4.5)
z
```

Out[76]:
```
7.5
```

You can return multiple values.

In [77]:
```
def get_sub_and_prod(x, y):
    sub = x - y
    prod = x * y
    return sub, prod

a, b = get_sub_and_prod(2, 3)
a, b
```

Out[77]:
```
(-1, 6)
```

Exercise

Write a function that converts a weight in grams to kilograms.

Use your function to convert 300 grams to kilograms.

In [ ]:

Exercise Key

Write a function that converts a weight in grams to kilograms.

Use your function to convert 300 grams to kilograms.

In [78]:
```
def g_to_kg(g):
    return g / 1000

print(f"300 g = {g_to_kg(300)} kg")
```

```
300 g = 0.3 kg
```

Function default arguments

In [79]:
```
def add_y_or_2(x, y=2):
    return x + y

add_y_or_2(3, 4)
```

Out[79]: 7

In [80]:
```
add_y_or_2(3)
```

Out[80]: 5

Function named arguments

In [81]:
```
def sub(x, y):
    return x - y

sub(3, 4)
```

Out[81]: -1

In [82]:
```
sub(x=3, y=4)
```

Out[82]: -1

In [83]:
```
sub(y=4, x=3)
```

Out[83]: -1

Variable scope

Variables defined at notebook level can be accessed from anywhere in the notebook including inside of all functions.

In [84]:
```
x = 3

print(x)
```

```
3
```

In [85]:
```
def what_is_x():
    print(x)
```

In [86]:
```
what_is_x()
```

```
3
```

Local scope within a function

Variables defined within a function only exist within the function.

In [87]:
```
x = 3

def setx():
    x = 2
    print(x)

setx()
```

```
2
```

What value does `x` have now?

In [88]:
```
x
```

Out[88]: 3

!!! Reassigning an input to a new local value does NOT affect the global input variable.

In [89]:
```python
x = 3

def changex(x):
    # Reassigning input `x` to new local value
    # does NOT affect global `x`
    x = x - 2
    print("local:", x)

changex(x)

print("global:", x)
```

```
local: 1
global: 3
```

!!! Mutating an input changes the global input variable.

In [90]:
```python
x = [1, 2, 3]

def changex(x):
    # Mutating input `x` changes global `x`
    x[1] = 8
    print("local:", x)

changex(x)

print("global:", x)
```

```
local: [1, 8, 3]
global: [1, 8, 3]
```

Classes

Learning goals

- You will be able to write and use your own classes.
- You will appreciate that classes are not always the best option.

A class is a template for a collection of data values (attributes) and functions (methods) that define some behaviors.

In [2]:
```python
class MySpikingNeuron:
```

```
    """ My cool neuron

    This neuron can spike
    and stuff!
    """

    def get_avg_seconds_to_next_spike(self):
        """ Returns the time in seconds to the next spike. """
        return 0.1
```

In [3]:

```
MySpikingNeuron.__dict__
```

Out[3]:

```
mappingproxy({'__module__': '__main__',
              '__doc__': ' My cool neuron\n    \n    This neuron can spike\
n    and stuff!\n    ',
              'get_avg_seconds_to_next_spike': <function __main__.MySpiking
Neuron.get_avg_seconds_to_next_spike(self)>,
              '__dict__': <attribute '__dict__' of 'MySpikingNeuron' object
s>,
              '__weakref__': <attribute '__weakref__' of 'MySpikingNeuron'
objects>})
```

To use the class you typically create an object that is an **instance** *of the class template*.

`neuron` is an instance of `MySpikingNeuron`

In [4]:

```
# instance = template()

neuron = MySpikingNeuron()

type(neuron)
```

Out[4]:

```
__main__.MySpikingNeuron
```

You can access the components of the class instance via `.component`.

In [5]:

```
neuron.get_avg_seconds_to_next_spike()
```

Out[5]:

```
0.1
```

Each class instance calls its `__init___(self)` method when it is created.

- `__init__` method is executed upon creation of each instance of a class.
- `self` refers to the instance of the class you are working with.

You use *self* to assign *attributes* to specific instances of a class.

In [6]:
```
class MySpikingNeuron:

    def __init__(self):
        self.spike_rate_per_sec = 20
```

In [7]:
```
neuron = MySpikingNeuron()  # __init__() called here with self = neuron
```

In [8]:
```
neuron.spike_rate_per_sec
```

Out[8]:
```
20
```

What is the reason for the following error?

In [9]:
```
class MySpikingNeuron:

    def __init__(self):
        spike_rate_per_sec = 20

neuron = MySpikingNeuron()
neuron.spike_rate_per_sec
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[9], line 7
      4         spike_rate_per_sec = 20
      6 neuron = MySpikingNeuron()
----> 7 neuron.spike_rate_per_sec

AttributeError: 'MySpikingNeuron' object has no attribute 'spike_rate_per_se
c'
```

`self` provides access to a specific instance of a class.

In [10]:
```
class MySpikingNeuron:

    def __init__(self):
        self.spike_rate_per_sec = 20

    def get_avg_seconds_to_next_spike(self):
        return 1 / self.spike_rate_per_sec
```

When calling a method of a class instance, `self` is *the instance itself*.

`neuron.get_avg_seconds_to_next_spike()` is equivalent to `MySpikingNeuron.get_avg_seconds_to_next_spike(self=neuron)`

In [11]:
```
neuron = MySpikingNeuron()

neuron.get_avg_seconds_to_next_spike()
```

Out[11]: 0.05

In [12]:
```
MySpikingNeuron.get_avg_seconds_to_next_spike(self=neuron)
```

Out[12]: 0.05

Classes are mutable.

In [13]:
```
class MySpikingNeuron:

    def __init__(self):
        self.spike_rate_per_sec = 20

    def get_avg_seconds_to_next_spike(self):
        return 1 / self.spike_rate_per_sec

neuron = MySpikingNeuron()
neuron.get_avg_seconds_to_next_spike()
```

Out[13]: 0.05

In [14]:
```
neuron.spike_rate_per_sec = 10
neuron.get_avg_seconds_to_next_spike()
```

Out[14]: 0.1

Editing an instance of a class does not affect other instances of the class and does not alter the class template.

In [15]:
```
another_neuron = MySpikingNeuron()

print(neuron.spike_rate_per_sec)
print(another_neuron.spike_rate_per_sec)
```

```
10
20
```

This way you can create many instances of `MySpikingNeuron` and assign them all different spike rates.

`__init__(self, ...)`

In [16]:
```
class MySpikingNeuron:

    def __init__(self, rate=20, region="hippocampus"):
        self.spike_rate_per_sec = rate
        self.region = region
```

In [17]:
```
neuronA = MySpikingNeuron(10, "cortex")
neuronB = MySpikingNeuron(5)
neuronC = MySpikingNeuron()
neuronD = MySpikingNeuron(region="spinal cord")
```

The special attribute `__dict__` is a dictionary of a class instance's attributes.

In [18]:
```
print(neuronA.__dict__)
print(neuronB.__dict__)
print(neuronC.__dict__)
print(neuronD.__dict__)
```

```
{'spike_rate_per_sec': 10, 'region': 'cortex'}
{'spike_rate_per_sec': 5, 'region': 'hippocampus'}
{'spike_rate_per_sec': 20, 'region': 'hippocampus'}
{'spike_rate_per_sec': 20, 'region': 'spinal cord'}
```

In [19]:
```
neuronA.spike_rate_per_sec, neuronB.spike_rate_per_sec
```

Out[19]:
```
(10, 5)
```

Another example of a class method.

In [20]:
```
class MySpikingNeuron:

    def __init__(self, rate=100, region=""):
        self.spike_rate_per_sec = rate
        self.brain_region = region

    def get_avg_seconds_to_next_spike(self):
        return 1 / self.spike_rate_per_sec

    def get_avg_spike_rate(self, another_neuron):
        return (self.spike_rate_per_sec + another_neuron.spike_rate_per_sec)
```

In [21]:
```
neuronA = MySpikingNeuron(10)
```

```python
neuronB = MySpikingNeuron(100)

neuronA.get_avg_spike_rate(neuronB)
```

Out[21]: 55.0

Class inheritance

In [22]:
```python
class MySpikingNeuron:

    def __init__(self):
        self.spike_rate_per_sec = 100

    def get_avg_seconds_to_next_spike(self):
        return 1 / self.spike_rate_per_sec
```

In [23]:
```python
class MyCoolSpikingNeuron(MySpikingNeuron):

    def __init__(self):
        # initialize the parent class
        MySpikingNeuron.__init__(self)

        self.coolness_factor = 9
```

In [24]:
```python
neuron = MyCoolSpikingNeuron()

neuron.__dict__
```

Out[24]: {'spike_rate_per_sec': 100, 'coolness_factor': 9}

In [25]:
```python
neuron.get_avg_seconds_to_next_spike()
```

Out[25]: 0.01

Let's check out a class that implements **linear regression** (*fitting a line to some data*): https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.line

The source code for this class is available at: https://github.com/scikit-learn/scikit-learn/blob/f3f51f9b6/sklearn/linear_model/_base.py#L529

In [26]:
```python
# You will find countless trivial examples of classes online such as:

class Dog:
```

```
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed
        self.tricks = ["sit"]

    def add_trick(self, trick):
        self.tricks.append(trick)

mydog = Dog("Scooter", "Husky")
mydog.add_trick("roll over")
```

Does this really warrant a *class*!?

What about a simple *Table* with columns for name, breed, tricks, etc.?

An opinion on classes and object-oriented programming (OOP):

In most cases a class is just an *unecessary* or *awkward* attempt at *compartmentalization*.

https://www.youtube.com/watch?v=QM1iUe6IofM&t=2618s

When is it appropriate to use a class?

Whenever it makes **A LOT OF SENSE** to keep *data (attributes)* and *functionality (methods)* together in a compartmentalized package.

Ummm... What does that mean?

There is *no singlular situation* in which a class is obviously the best choice. So instead, how about an **example**:

Let's check out a class that implements **linear regression** (*fitting a line to some data*): https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.line

The source code for this class is available at: https://github.com/scikit-learn/scikit-learn/blob/f3f51f9b6/sklearn/linear_model/_base.py#L529

Do you think it was useful to implement `LinearRegression` as a class?

Imagine that we implement classes for multiple analysis methods such as `LinearRegression`, `RidgeRegression`, `RandomForestClassification`, etc.

and each of them has a `.fit(data)` method.

Then we can write code for many different types of analyses that has a *consistent form*:

```
model = LinearRegression()
model.fit(data)

model = RidgeRegression()
model.fit(data)

model = RandomForestClassification()
model.fit(data)
```

Modules

Learning goals

- You will be able to organize your code into your own modules.

A **module** is just a **collection of any kind of objects** including *data*, *functions*, *classes*, etc.

That's pretty much the same thing as a class!?

But a *module is a single python* `.py` *file*, whereas a *class is just a code block* within a file.

You can think of it as:

- **module** = compartmentalization on a *macro scale*
- **class** = compartmentalization on a *micro scale*

Classes are only a good idea in *LIMITED CASES*.

Modules are almost *ALWAYS A GOOD IDEA* if your code contains more than a few very short functions or classes.

Module `MyNeuron.py` contains a defintion for the class `MySpikingNeuron`.

In [27]:
```
import MyNeuron

neuronA = MyNeuron.MySpikingNeuron(10)
neuronB = MyNeuron.MySpikingNeuron(100)
```

In [28]:
```
import MyNeuron as mn

neuronA = mn.MySpikingNeuron(10)
neuronB = mn.MySpikingNeuron(100)
```

In [29]:
```
from MyNeuron import MySpikingNeuron

neuronA = MySpikingNeuron(10)
neuronB = MySpikingNeuron(100)

MySpikingNeuron
```

Out[29]:
```
MyNeuron.MySpikingNeuron
```

You have access to everything in the module.

In [30]:
```
import MyNeuron2
```

In [31]:
```
MyNeuron2.brain_region
```

Out[31]:
```
'hippocampus'
```

In [32]:
```
neurons = MyNeuron2.create_three_random_neurons()

neurons
```

Out[32]:
```
[<MyNeuron2.MySpikingNeuron at 0x10a4dc5d0>,
 <MyNeuron2.MySpikingNeuron at 0x10b1a2410>,
 <MyNeuron2.MySpikingNeuron at 0x10b825f90>]
```

In [33]:
```
for neuron in neurons:
    print(neuron.spike_rate_per_sec)
```

```
21.971658452882807
11.48211918781041
10.762747366938804
```

Nested modules

`MyNeuron3` does not directly contain the definition for the `MySpikingNeuron` class.

Instead, `MyNeuron3` itself imports `MyNeuron`.

In [34]:
```
import MyNeuron3
```

In [35]:
```
neuron = MyNeuron3.MySpikingNeuron(10)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[35], line 1
----> 1 neuron = MyNeuron3.MySpikingNeuron(10)

AttributeError: module 'MyNeuron3' has no attribute 'MySpikingNeuron'
```

In [36]:

```
neuron = MyNeuron3.MyNeuron.MySpikingNeuron(10)
neuron
```

Out[36]:

```
<MyNeuron.MySpikingNeuron at 0x10b1a3c50>
```

In [37]:

```
neurons = MyNeuron3.create_three_random_neurons()

neurons
```

Out[37]:

```
[<MyNeuron.MySpikingNeuron at 0x10b84efd0>,
 <MyNeuron.MySpikingNeuron at 0x10b84f450>,
 <MyNeuron.MySpikingNeuron at 0x10b84f0d0>]
```