Week 1 - Syntax, data types, and built-in functions

Before we dive into the basics, let's remind ourselves that Python is an object-oriented programming language, and that virtually everything in Python is an "object" that bundles together some data and functionality.

When used in a programming sense, the concept of an "object" is abstract. We can't physically reach out to the object, pick it up, and manipulate it in the same way that we might a pen or a pencil. But the abstraction make sense, because the objects of computer programs typically represent objects in the real world.

To illustrate the point, let's make a `Pen` object and use it write "Hello, World!".

In [1]:

```
# Define the Pen class
class Pen:
    def __init__(self):
        # Attribute to keep track of ink
        self.ink_left = 100

    # Method for writing stuff
    def write_something(self, what):
        if len(what) > self.ink_left:
            print("Not enough ink!")
        else:
            print(what)
            self.ink_left = self.ink_left - len(what)

# Create an instance of Pen
my_pen = Pen()

# Call the write_something method
my_pen.write_something(what="Hello, World!")

# Check ink left
print(f"my_pen has {my_pen.ink_left} ink left")
```

```
Hello, World!
my_pen has 87 ink left
```

In the above, `class` is used to define a type of object called `Pen`, which has an `ink_left` *attribute* and a `write_something` *method*. We then create an *instance* of `Pen` and use `write_something` to print the message "Hello, World!". Every time `write_something` is called, the ink gets depleted, and if there isn't enough ink to write the message, the program will let us know.

This is a glimpse into Python's `class` mechanism, which is what provides the object-oriented features to the language. However, Python does not enforce the object-oriented style, which means that for now we need not worry about `class`, `__init__`, `self`, etc.

1. Syntax

In the context of programming languages, *syntax* refers to the rules that govern the structure of statements and expressions.

One of Python's most attractive features is its clear, concise and readable syntax. We will learn more about this continuously throughout the course, but for now let's explore some of the main components.

1.1 Comments

Comments are bits of text that exist inside your code, but which do not get evaluated when the code is executed. You can use them to explain or annotate a piece of code that you have written, so when you (or another developer) reads it at a later stage it will be clear what the code is doing, or why it was written in that particular way.

In Python, comments are written by putting a `#` (hash) symbol before the comment text. Anything that comes after a `#` on a particular line of code is ignored.

In [6]:
```
# This is a single line comment
```

In [7]:
```
name = "Fred"  # This is an inline comment
```

In [8]:
```
# This is a multiline comment
# that requires more than one line
# for it to be complete
```

Remember the `"Hello, World!"` program from last week? Here it is again with a single line comment to explain what it does.

In [27]:
```
# Print the text "Hello, World!" to the console
print("Hello, world!")
```

```
Hello, world!
```

Always comment your code, preferably with plain and succinct language. Your colleagues and your future self will be thankful!

1.2 Variables

In Python, variables are essentially names attached to objects.

To put it more accurately, Python variables hold a reference or "pointer" to an address in computer memory where an object is stored, but it isn't particularly helpful to think of variables in this way when learning the language.

Just think of variables as "names for objects" or "boxes to put things in" and you won't go far wrong.

Values are assigned to variables using the `=` (equals) operator. The value goes on the right, and the variable name goes on the left.

`<variable_name> = <value>`

In [59]:
```
# Assign the value "Fred" to the variable "first_name"
first_name = "Fred"

# Assign the value "Jones" to the variable 'second_name'
second_name = "Jones"

# Assign the value 42 to the variable 'age'
```

```
age = 42

# Print out his full name and age
print(f"His name is {first_name} {second_name} and he is {age} years old")
```

```
His name is Fred Jones and he is 42 years old
```

Variable names should be descriptive enough to allow another person reading your code to have a guess at the type of data the variable contains, so think carefully before putting a name to a variable! In the above code, `first_name` and `second_name` are good descriptive variable names, and it would not be surprising to find out their values are "Fred" and "Jones".

When naming variables, you can use uppercase `[A-Z]` and lower case `[a-z]` letters, digits `[0-9]`, and the `_` character, but **variable names can not start with a digit!** See what happens if you run the code below.

In [142…
```
Variable_Name_123 = "something"  # A valid variable name
print(Variable_Name_123)
```

```
something
```

In [143…
```
123_Variable_Name = "something else"  # Not a valid variable name
print(123_Variable_Name)
```

```
  Input In [143]
    123_Variable_Name = "something else"  # Not a valid variable name
       ^
SyntaxError: invalid decimal literal
```

We get a `SyntaxError`, because we violated Python's syntax rules by attempting to use digits at the start of a variable name. Errors like this can be daunting, but learning to interpret them is a key aspect of developing your Python programming abilities. When I see an error I don't understand, the first thing I do is Google it. In this case, I Googled "*SyntaxError: invalid decimal literal*", and the first search result explained everything!

Note that using **`lower_case_with_underscore`** variable names (sometimes called "snake_case") is a widely adopted standard in Python. You don't have to name your variables in this style, but I personally find it more clean and readable than alternate naming conventions (e.g., "CamelCase"). For more guidelines and recommendations on coding style in Python, PEP8 is the definitive source.

A final note on variable names... They are case sensitive! So, `txt`, `Txt` and `TXT` can each be used as variable names for different values.

In [62]:
```
txt = 'One'
Txt = 'Two'
TXT = 'Three'
print(txt, Txt, TXT)
```

```
One Two Three
```

1.3 Keywords

Python has a set of keywords which have their own specific purpose or meaning within the language. They can not be used as variable names, function names, or as any other form of identifier.

We can get get a list of the available keywords by typing:

In [11]:
```
help("keywords")
```

```
Here is a list of the Python keywords.  Enter any keyword to get more help.

False               break               for                 not
None                class               from                or
True                continue            global              pass
__peg_parser__      def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
```

Incidentally, we have just introduced one of the most important functions in the Python programming language: `help()`.

`help()` will retrieve and print any available documentation for whichever *object* is passed to it. For example, if I wanted to learn more about the `print` statement, I could say:

In [15]:
```
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

In this course we will encounter many, but not all, of Python's keywords. For now, you could try asking for help on the keywords that were emboldened in the presentation slides. For example:

In [69]:
```
help('if')
```

```
The "if" statement
******************

The "if" statement is used for conditional execution:

   if_stmt ::= "if" assignment_expression ":" suite
               ("elif" assignment_expression ":" suite)*
               ["else" ":" suite]

It selects exactly one of the suites by evaluating the expressions one
by one until one is found to be true (see section Boolean operations
for the definition of true and false); then that suite is executed
(and no other part of the "if" statement is executed or evaluated).
If all expressions are false, the suite of the "else" clause, if
present, is executed.

Related help topics: TRUTHVALUE
```

Note that if you call `help()` without any arguments, you may find yourself in an *interactive* help session, which you can exit by typing `quit` and hitting `Enter`.

1.4 Lines, indentation, and whitespace

Many computer languages don't care about whitespace, but Python does!

In Python, whitespace is part of the syntax. Programs are initially divided into logical lines, and the whitespace is then used by the Python interpreter to parse code into syntactic blocks.

For example, when writing a loop or defining a function, one must indent subsequent lines with 4 spaces (a `Tab` may also be used, but this is generally discouraged).

In [184…
```
world_is_round = True

if world_is_round:
    # 4 spaces indentation
    print("The world is round")
```

```
The world is round
```

If we repeat this code without the indentation, we get an `IndentationError`

In [185…
```
world_is_round = True

if world_is_round:
print("The world is round")
```

```
  Input In [185]
    print("The world is round")
    ^
IndentationError: expected an indented block
```

Note that whitespace within a line is ignored.

In [186…
```
world_is_round =      True

if    world_is_round    :
    print(     "The world is round"   )
```

```
The world is round
```

But you should never use whitespace like this as it makes the code hard to read. PEP8 has recommendations on what it considers to be the correct use of whitespace. The take-home message? Use it sparingly and in a consistent manner.

2. Built-in data types

We already know that everything in Python is an object. Now, consider that **every object has a type**.

Python has a range of built-in data types, but some are more common than others. For now we will focus on numbers (`int`, `float`), booleans (`True`, `False`), sequences (`str`, `list`, `tuple`), and dictionaries (`dict`).

N.B. The type of an object can always be revealed using Python's built-in `type()` function.

2.1 Numbers

Python distinguishes between three types of number: integer, floating point, and complex. We will focus on integers and floating points (because complex numbers are complex and I don't know much about them)!

Integers (e.g., `42`, `36`, `-5`) are whole numbers, and have type `int`.

In [231…
```
# Integer
num_int = 42

print(f"The type of num_int is {type(num_int)}")
print(f"The value of num_int is {num_int}")
```

```
The type of num_int is <class 'int'>
The value of num_int is 42
```

Floating point numbers (e.g., `3.142`, `365.25`, `-.06`) are numbers with a fractional component, and have the type `float`.

In [232…
```
# Float
num_float = 365.25

print(f"The type of num_float is {type(num_float)}")
print(f"The value of num_float is {num_float}")
```

```
The type of num_float is <class 'float'>
The value of num_float is 365.25
```

We can use the `int()` and `float()` functions to convert specified values to the respective type.

In [4]:
```
int(num_float)
```

Out[4]:
```
365
```

In [5]:
```
float(num_int)
```

Out[5]:
```
42.0
```

And we can even convert string representations of numbers.

In [9]:
```
int('42')
```

Out[9]:
```
42
```

In [10]:
```
float('365.25')
```

Out[10]:
```
365.25
```

At a very basic level, Python functions as a calculator. Addition, subtraction, multiplication and division are accomplished with the `+`, `-`, `*` and `/` symbols, known as *operators*.

In [11]:
```
5 + 9
```

Out[11]:
```
14
```

In [12]:
```
10 - 5
```

Out[12]: 5

In [13]:
```
2 * 2
```

Out[13]: 4

In [14]:
```
10 / 5
```

Out[14]: 2.0

We can also calculate the power of a number using the `**` (double asterix) operator.

In [15]:
```
2 ** 4
```

Out[15]: 16

2.2 Booleans

The `boole` data type, named after George Boole (1815-1864), has two possible values: `True` and `False`.

In [233…
```
type(False)
```

Out[233]: bool

In [234…
```
type(True)
```

Out[234]: bool

Booleans are used to represent the truth of an expression in conditional statements.

In [3]:
```
world_is_round = True  # Boolean expression

if world_is_round:  # Conditional statement
    print("The world is round")  # Executed only if expression is True
```

```
The world is round
```

What will happen when you run the cell below? Give it a try!

In [4]:
```
world_is_flat = False

if world_is_flat:
    print("The world is flat")
```

The Booelean operators `and`, `or`, and `not`, can be used to form conditional statements (more on this next week).

In [34]:
```
# Some Boolean expressions
world_is_round = True
world_is_flat = False
cake_is_yummy = True
life_is_good = True

# The 'and' operator returns True if all operands evaluate as True
if cake_is_yummy and life_is_good and world_is_round:
    print("Cake is yummy, life is good, the world is round")
```

```
# The 'or' operator returns True if at least
# one of the operands evaluates as True
if world_is_flat or cake_is_yummy:
    print("The world is flat or cake is yummy (at least one is true)")

# The 'not' operator returns True if its operand evaluates as False,
# otherwise it returns False
if not world_is_flat:
    print("The world is not flat")
```

```
Cake is yummy, life is good, the world is round
The world is flat or cake is yummy (at least one is true)
The world is not flat
```

Interestingly, `boole` is a subclass of `int`, which means `True` and `False` behave as integers in an arithmetic sense.

In [157…:
```
True + True
```

Out[157]:
```
2
```

In [36]:
```
(True * 10) / (True + True)
```

Out[36]:
```
5.0
```

This means that you can use `0` and `1` in place of `False` and `True`.

In [6]:
```
if 0:  # False
    print('This will not get printed')

if 1:  # True
    print('This will get printed')
```

```
This will get printed
```

2.3 Strings

Besides numbers, Python can manipulate strings, which are sequences of values represented by Unicode points.

In [193…:
```
my_string = 'something'
type(my_string)
```

Out[193]:
```
str
```

Strings can be enclosed in single quotes (`'...'`) or double quotes (`"..."`) with the same result. Once a string has been created, it can not be manipulated, which is to say that **strings are immutable**.

In [42]:
```
print('Fred')
print("Fred")
```

```
Fred
Fred
```

It is also possible to use the `\` (backslash) character to *escape* quotes that are needed within a string.

In [43]:
```
print("\"Python\" is Fred\'s favourite word.")
```

```
"Python" is Fred's favourite word.
```

In fact, there are quite a few more special characters involving the backslash. The ones in bold are worth remembering.

- **`\n` - Newline**
- **`\t` - Horizontal tab**
- `\r` - Carriage return
- `\b` - Backspace
- `\f` - Form feed
- **`\'` - Single quote**
- **`\"` - Double quote**
- **`\\` - Backslash**
- `\v` - Vertical tab
- **`\N` - N is the number for Unicode character**
- `\NNN` - NNN is digits for Octal value
- `\xNN` - NN is a hex value; \x is used to denote following is a hex value.
- `\a` - Bell sound, actually default chime

`\n` and `\t` are useful as they enable you to include new lines and tabs in a string.

In [38]:
```python
print('TODO:\n\t1. Clean kitchen\n\t2. Post letter\n\t3. Hang out washing')
```

```
TODO:
	1. Clean kitchen
	2. Post letter
	3. Hang out washing
```

And there is much fun to be had with Unicode! I started learning Greek a few years back. Here's one of my favorite Greek proverbs.

In [73]:
```python
print('\u03A4\u03B1 \u03C0\u03BF\u03BB\u03BB\u03AC \u03BB\u03CC\u03B3\u03B9\u03B1 \u03B5
```

```
Τα πολλά λόγια είναι φτώχια
```

We can also print emojis. Here's 100 snakes!

In [71]:
```python
print('\U0001F40D'*100)
```

```
🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍
🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍
🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍
```

In [66]:
```python
print('\U0001F642')
```

```
🙂
```

Let's move on... Strings have many useful methods that can be accessed via Python's dot notation. For example, the `.upper()` method will convert all characters to their upper case.

In [240…
```python
msg = "urgent: this message requires your attention"
print(msg)
print(msg.upper())
```

```
urgent: this message requires your attention
URGENT: THIS MESSAGE REQUIRES YOUR ATTENTION
```

As with all Python objects, we can see what methods are available by typing a `.` after the variable and pressing `tab`. Then, if you want to learn more about a particular method, you can use the `help()` function.

In [241…
```
help(msg.upper)
```

```
Help on built-in function upper:

upper() method of builtins.str instance
    Return a copy of the string converted to uppercase.
```

N.B. A more convenient way to get help when working in an IPython console is by simply putting a question mark after the object

In [238…
```
# IPython introspection with question mark
msg.upper?
```

```
Signature: msg.upper()
Docstring: Return a copy of the string converted to uppercase.
Type:      builtin_function_or_method
```

Here are a few more examples of string methods in action.

In [242…
```
# Split the string into a list of separate words
print(msg.split())

# Count the number of occurences of 'o'
print(msg.count('o'))

# Replace substring and then convert to uppercase by 'chaining' methods together
print(msg.replace('attention', 'immediate attention').upper())
```

```
['urgent:', 'this', 'message', 'requires', 'your', 'attention']
2
URGENT: THIS MESSAGE REQUIRES YOUR IMMEDIATE ATTENTION
```

Try using the cell below to create a string variable and explore string methods!

In [110…
```
# Define a string variable here!
```

In Python there are various ways to *format* strings, one of which is by calling the `.format()` method. This method will substitute curly braces `{}` for the values that are specified in the call to the method.

In [70]:
```
first_name = 'Fred'
last_name = 'Jones'
print('His name is {} {}'.format(first_name, last_name))
```

```
His name is Fred Jones
```

Since Python 3.6, we can format strings using the f-String syntax, which is my personal preference. f-Strings are prefixed with an `f` and the substitution values are placed directly in the curly braces.

In [73]:
```
print(f'His name is {first_name} {last_name}')
```

```
His name is Fred Jones
```

Previously we saw that the `+` and `*` operators can be used for addition and multiplication in a numerical context. These particular operators can also be applied to strings!

In [92]:
```
# Use the + operator to concatenate (i.e. glue together) strings
```

```
'I ' + 'am ' 'the ' + 'Cookie ' + 'Monster ' + 'and ' + 'I ' + 'love'
```

```
Out[92]: 'I am the Cookie Monster and I love'
```

```
In [111…  # Use the * operator to create multiple copies of (i.e. repeat) a string
          'COOKIES ' * 10
```

```
Out[111]: 'COOKIES COOKIES COOKIES COOKIES COOKIES COOKIES COOKIES COOKIES COOKIES COOKIES '
```

Finally (though not exhaustively), strings are sequences of data, which means they can be *indexed* and *sliced* using square brackets `[]` and numerical indices. Recall that indexing starts at `0` in Python, so to get the first character of a string variable we would use `[0]`

```
In [112…  msg = 'supercalifragilisticexpialidocious'
          # Get the first character from the string
          msg[0]
```

```
Out[112]: 's'
```

```
In [113…  # Get characters from 0 to 5
          msg[0:5]
```

```
Out[113]: 'super'
```

2.4 Lists

Python's `list` is a versatile, compound data type used for grouping together other values, which may or may not have the same type (although items in a list typically have the same type).

Lists are defined with square brackets and commas between each item.

```
In [86]:  my_shopping_list = ['bread', 'milk', 'soap', 'tin foil', 'dishwasher tablets']
          type(my_shopping_list)
```

```
Out[86]: list
```

```
In [87]:  print(my_shopping_list)
```

```
['bread', 'milk', 'soap', 'tin foil', 'dishwasher tablets']
```

As with strings, lists support indexing and slicing via square brackets and numerical indices.

```
In [88]:  # Get the first item in the list
          my_shopping_list[0]
```

```
Out[88]: 'bread'
```

```
In [89]:  # Get items 1 to 3 in the list
          my_shopping_list[1:3]
```

```
Out[89]: ['milk', 'soap']
```

However, unlike strings, **lists are mutable**, which means elements can be removed or reassigned.

```
In [90]:  # Change 'bread' to 'sourdough'
          my_shopping_list[0] = 'sourdough'
          print(my_shopping_list)
```

```
['sourdough', 'milk', 'soap', 'tin foil', 'dishwasher tablets']
```

In [91]:
```
# Use the del (delete) statement to remove 'tin foil' from the list
del my_shopping_list[3]
print(my_shopping_list)
```

```
['sourdough', 'milk', 'soap', 'dishwasher tablets']
```

Concatenation and repeating with the `+` and `*` operators works the same as with strings.

In [92]:
```
your_shopping_list = ['flour', 'butter', 'eggs']

# Concatenate two lists with the + operator
combined_list = my_shopping_list + your_shopping_list
print(combined_list)
```

```
['sourdough', 'milk', 'soap', 'dishwasher tablets', 'flour', 'butter', 'eggs']
```

In [93]:
```
# Repeat the list with the * operator
duplicated_list = combined_list * 2
print(duplicated_list)
```

```
['sourdough', 'milk', 'soap', 'dishwasher tablets', 'flour', 'butter', 'eggs', 'sourdoug
h', 'milk', 'soap', 'dishwasher tablets', 'flour', 'butter', 'eggs']
```

Lists also have useful methods that can be accessed via the `.` notation. Here are a few examples.

In [94]:
```
# Print the current shopping list
print(my_shopping_list)

# Reverse the list
my_shopping_list.reverse()
print(my_shopping_list)

# Append an item to the end of the list
my_shopping_list.append('apples')
print(my_shopping_list)

# Sort list alphabetically
my_shopping_list.sort()
print(my_shopping_list)
```

```
['sourdough', 'milk', 'soap', 'dishwasher tablets']
['dishwasher tablets', 'soap', 'milk', 'sourdough']
['dishwasher tablets', 'soap', 'milk', 'sourdough', 'apples']
['apples', 'dishwasher tablets', 'milk', 'soap', 'sourdough']
```

Try creating a list in the cell below and explore the list methods! Remember, if you are unsure what a method does you can use the `help()` function.

In [ ]:
```
# Create a list in this cell!
```

2.5 Tuples

Tuples are just like lists, except they are **immutable**, which means they can't be changed after they've been made.

It may seem strange to have two separate data types with only this small difference between them, but the versatility of lists comes at the expense of more allocated computer memory. If you know that a sequence of values is not going to change across the life of a computer program, using a tuple instead of a list can improve performance.

```
In [97]: # Latitude and longitude of York Minster
         location = (53.96228434988206, -1.081881847036603)
         type(location)

Out[97]: tuple
```

Indexing of tuples works the same. as with lists and strings

```
In [99]: latitude = location[0]  # First item in the tuple
         latitude

Out[99]: 53.96228434988206
```

```
In [100… longitude = location[1]  # Second item in the tuple
         longitude

Out[100]: -1.081881847036603
```

There is little else to say about tuples, other than they have two built-in methods: `.count()` and `.index()`. Try defining a tuple in the cell below and learn more about these methods!

```
In [ ]: # Define a tuple in this cell
```

2.6 Dictionaries

Dictionaries, like lists, are a mutable data type for storing collections of objects. The difference with dictionaries is that **data are stored in key-value pairs**.

We define dictionaries using curly braces, commas to separate key-value pairs, and colons to separate keys and values.

`my_dict = {key: value, key: value, ..., key: value}`

Here's an example dictionary that stores some data about the University of York.

```
In [119… # An example dictionary
         university = {
             'name': 'University of York',
             'location': (53.946216224889824, -1.0516966993211523),
             'contact': {
                 'tel': '+44 (0)1904 32 0000',
                 'fax': '+44 (0)1904 32 3433'
             },
             'website': 'https://www.york.ac.uk/',
             'founded': 1963,
             'russel_group': True,

         }
         type(university)

Out[119]: dict
```

To access a value from a dictionary, we simply use the key as an index.

```
In [106…  university['name']
```

```
Out[106]: 'University of York'
```

In the above example above, all of the dictionary keys are of string type. However, we can use any hashable (essentially synonymous with mutable) data type as a dictionary key.

```
In [140…  my_dict = {
              1: 'has a integer key',
              2.0: 'has a floating point key',
              (3, 4.0): 'has a tuple key',
              'five': 'has a string key'
          }
```

```
In [141…  my_dict[(3, 4.0)]
```

```
Out[141]: 'has a tuple key'
```

The only other important restriction on dictionary keys is that there can be no duplicates!

As with lists, strings, and tuples, dictionaries have some useful built-in methods. Here are some of them in action.

```
In [130…  # Use .keys() to get a list of a dictionary's keys
          print(university.keys())
```

```
dict_keys(['name', 'location', 'contact', 'website', 'founded', 'russel_group'])
```

```
In [137…  # Use .values() to get a list of a dictionary's values
          print(university.values())
```

```
dict_values(['University of York', (53.946216224889824, -1.0516966993211523), {'tel': '+
44 (0)1904 32 0000', 'fax': '+44 (0)1904 32 3433'}, 'https://www.york.ac.uk/', 1963, Tru
e])
```

```
In [133…  # Use .items() to get a list of a dictionary's items (i.e., key:value pairs)
          print(university.items())
```

```
dict_items([('name', 'University of York'), ('location', (53.946216224889824, -1.0516966
993211523)), ('contact', {'tel': '+44 (0)1904 32 0000', 'fax': '+44 (0)1904 32 3433'}),
('website', 'https://www.york.ac.uk/'), ('founded', 1963), ('russel_group', True)])
```

Try defining your own dictionary in the cell below and experiment with dictionary methods!

```
In [ ]:  # Define a dictionary here!
```

3. Built-in Functions

Python has some built-in functions that are always available for use. Some are more obscure than others and tend to be less relevant for beginners. Up to now, the built-in functions we have encountered include `help()`, `print()`, `input()`, `type()`, `int()`, `float()`, `bool()`, `str()`, `list()`, `tuple()`, and `dict()`.

Use the cell below to refresh your memory on these. You can use the `help()` function, or type the name of the object (without the `()`) followed by a question mark.

```
In [223…  # Use this cell to get help on built-in functions
```

```
int?
```

```
Init signature: int(self, /, *args, **kwargs)
Docstring:
int([x]) -> integer
int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments
are given.  If x is a number, return x.__int__().  For floating point
numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string,
bytes, or bytearray instance representing an integer literal in the
given base.  The literal can be preceded by '+' or '-' and be surrounded
by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
Base 0 means to interpret the base from the string as an integer literal.
>>> int('0b100', base=0)
4
Type:           type
Subclasses:     bool, IntEnum, IntFlag, _NamedIntConstant
```

Another very useful built-in function is `len()`, which returns the number of items in a sequence. The `Pen` example at the start of this notebook uses `len()` to deplete its `ink_left` attribute according to the number of characters in the message printed `.write_something()`. Take another look, hopefully it will make a bit more sense now!

Here's a look at `len()` in action.

In [264…
```
# Use of len() to get the number of characters in a string
msg = 'hello'
len(msg)
```

Out[264]:
```
5
```

In [265…
```
# Use of len() to get the number of items in a list
shopping_list = ['bread', 'milk', 'tin foil']
len(shopping_list)
```

Out[265]:
```
3
```

Some other useful built-ins for working with numerical data are `min()`, `max()`, `sum()`, and `round()`. Here's a quick look at each of them.

In [1]:
```
# A list of numbers
numbers = [2, 55, 67, 34.07, 900]
```

In [2]:
```
# min() returns the smallest item
min(numbers)
```

Out[2]:
```
2
```

In [3]:
```
# max() returns the biggest item
max(numbers)
```

Out[3]:
```
900
```

In [4]:
```
# sum() returns the sum of numeric values
sum(numbers)
```

```
Out[4]: 1058.07
```

```
In [5]: # round() rounds a number to a specified precision
        pi = 3.14159265359
        round(pi, 3)
```

```
Out[5]: 3.142
```

4. That's it for now!

```
In [10]: print('Time for a \u2615?')
```

```
Time for a ☕?
```