Getting Started: Fundamentals

Introduction to Python

Data Sciences Institute, University of Toronto
Instructor: A Mahfouz | TA: Kaylie Lau
July 2022

Contents:

1. Python Fundamentals
    A. Types
    B. Statements and Expressions
    C. Logical Data
    D. Variables
    E. Errors
    F. Comments and Readability
    G. Functions
    H. Strings
    I. Input

Programming

What is programming ?

A *program* is a set of instructions. *Programming* defines new sets of instructions to perform a useful task. A *programming language* lets us write those instructions in a way that a computer can interpret and execute.

Python

What is Python?

Python is a general-purpose programming language first released in 1991. It has since become a popular language for data science, thanks to an enthusiastic community and a large ecosystem of code libraries and tools that make it easier to perform common tasks throughout the data science life cycle.

Setup

Options

Google Colab	Anaconda
• Easy to get started • Less flexible • Runs online through a Google account	• Can be tricky to install • Lots of flexibility • Installed on your own computer
Go to **https://colab.research.google.com/** In the upper left corner, click **File**, then **New notebook**.	Download and install from **https://www.anaconda.com/products/individual#Downloads** Open **Anaconda Navigator**. Then, install and launch **Jupyter Notebook.**

Anaconda and Notebooks

Anaconda is a software suite that consists of Python and several complementary data science applications. It includes a Python *interpreter*, or a program that translates Python code into machine instructions. It also includes *Jupyter Notebook*, a web application that lets us combine pieces of executable code, text, images, and visualizations in a single document, or notebook. Even though Jupyter Notebook is a web application, notebooks are stored locally on your computer, not hosted online by default.
Sections of a notebook are called *cells*. Cells can be run independently of each other, and in any order.

Google Colab

Google Colab is a Jupyter Notebook environment hosted on the cloud -- that is, on Google's servers. Google Colab provides a standardized notebook environment, with common data science libraries already installed.

Python Fundamentals

Types

Every piece of data in Python has a *type*. Data types dictate how values are stored by the computer and what operations we can perform on the data. For example, an `int`, or integer, value must be a whole number, and we can use it for arithmetic. The `float`, or floating point data type supports decimal numbers. There are other types for text (`string`), true/false values (`bool`), and more complex data that we will go into later.

Statements and Expressions

A *statement* in programming is a command. You can think of them as the smallest units of code. An *expression* is a statement that produces a single value.
We can use arithmetic operators like `+` and `-` to write simple expressions.

```
In [1]: 5 + 6
```

```
Out[1]:
11
```

`5 + 6` is an expression.

`+` is an operator.

`5` and `6` are operands.

`11` is a value.

The result of an expression may be a different data type than its inputs.

```
In [2]: 5 / 2
```

```
Out[2]:
2.5
```

Practice

What values do the following expressions produce?

```
9 - 3
8 * 2.5
9 / 2
9 / -2
```

Additional Arithmetic Operators

Operator	Description
//	Integer division (always rounds down)
%	Modulo or remainder
**	Exponentiation

Examples

```
In [4]: 10 // 3
```

```
Out[4]:
3
```

```
In [5]: 10 % 3
```

```
Out[5]:
1
```

```
In [6]: 10 / 3
```

```
Out[6]:
3.3333333333333335
```

```
In [7]: 10 ** 3
```

```
Out[7]:
1000
```

Logical Data

True/False

Python also has a data type, `bool`, for `True` and `False` values. The term `bool` derives from Boolean, which is itself a reference to the mathematician and logician George Boole, whose work underpins information and computing.

Comparison operators

Python has comparison operators as well as arithmetic operators. Unlike arithmetic expressions, which produce integer or float values, comparison expressions produce a Boolean value.

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

Examples

```
In [8]: 50 > 25
```

```
Out[8]:
True
```

```
In [9]: -15 >= -14.99
```

```
Out[9]:
False
```

```
In [10]: 20 == 20
```

```
Out[10]:
True
```

Operator precedence

Arithmetic and comparison operators are evaluated in the following order. We can enclose operations in parentheses to override the order of precedence -- operations in parentheses are evaluated before the rest of the expression.

Order	Operator	Description
1	`**`	Exponentiation
2	`-`	Negation
3	`*`, `/`, `//`, `%`	Multiplication, division, integer division, and modulo
4	`+`, `-`	Addition and subtraction
5	`<`, `<=`, `>`, `>=`, `==`, `!=`	Less than, less than or equal to, greater than, greater than or equal to, equal, not equal

Practice

In which order will the expressions be evaluated?

```
6 * 3 + 7 * 4
5 + 3 / 4
5 - 2 * 3 ** 4
(5 - 2) * 3 ** 4
5 + 2 >= 3 * 4
```

Variables

What is a variable?

A *variable* is a name that refers to a value. Variables make it easier to keep track of data. Variable names in Python can include letters, digits, and underscores. They cannot start with a digit. They are also case sensitive, so `variable` and `Variable` would be two different variables!

Creating variables

To create a variable, we assign it a value using the assignment operator `=`.

```
In [11]: degrees_celsius = 25
```

This statement is called an *assignment statement*. Now that the variable `degrees_celsius` has been assigned a value, we can use it anywhere we would use that value.

```
In [12]: degrees_fahrenheit = (9 / 5) * degrees_celsius + 32
         degrees_fahrenheit
```

```
Out[12]:
77.0
```

Reassigning variables

When we reassign a variable, we change the value that variable refers to. Reassigning a variable does not change any other variable. Notice that `degrees_fahrenheit` stayed the same, even when we reassigned `degrees_celsius`.

In [13]:
```
degrees_celsius = 0
degrees_fahrenheit
```

Out[13]:

```
77.0
```

We can use a variable on both sides of an assignment statement. This is useful for updating a variable based on its current value.

In [14]:
```
degrees_celsius = degrees_celsius + 10
degrees_celsius
```

Out[14]:

```
10
```

Notice what happens when we set one variable equal to another, then reassign the first one.

In [15]:
```
a = 1
b = a
b
```

Out[15]:

```
1
```

In [16]:
```
a = 2
b
```

Out[16]:

```
1
```

Augmented assignment

While `degrees_celsius = degrees_celsius + 10` is a valid statement, there is a more concise way to express this: by putting the operator `+` before the assignment operator `=`.

```
In [18]: degrees_celsius = 0
         degrees_celsius += 10
degrees_celsius
```

```
Out[18]:
```

```
10
```

Similarly, subtraction, multiplication, division, expontiation, integer division, and modulo can be used as an augmented assignment.

Errors

What causes errors?

Errors indicate that there is something in the code that the computer cannot execute. We will see some of the reasons that can happen. When an error occurs, Python produces an error message called a *traceback*. Tracebacks can be quite cryptic, but they tell us what code was running at the time of the error. When in doubt, copying the error message at the bottom of a traceback into a search engine can help.

Syntax errors

A *syntax error* occurs when our program contains code that isn't valid Python. Below, we are trying to assign the value `x` to the variable `12` -- but variable names that start with digits are not allowed in Python!

```
In [19]: 12 = x
```

```
  Input In [19]
    12 = x
    ^
SyntaxError: cannot assign to literal here. Maybe you meant '==' instead of '='?
```

```
In [20]: 25 -
```

```
  Input In [20]
    25 -
         ^
SyntaxError: invalid syntax
```

Above, Python expects another operand to the right of `-`, and throws a syntax error because there isn't a second operand.

Name errors

A *name error* occurs when we try to use a variable that hasn't been assigned yet. Name errors are especially common when working in notebooks, where code is often not run from top to bottom.

```
In [21]: my_variable + 1
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [21], in <cell line: 1>()
----> 1 my_variable + 1

NameError: name 'my_variable' is not defined
```

Practice

Which of these expressions results in a SyntaxError?

```
((((5 * 4 ** 7))))

84 *= 0.5 / 7

(-(-(-(-5 * (4 + 3)))))

5 * 3 = weight

`73 / -----------5
```

Comments

What are comments?

Comments are a way to annotate code within code. They're used to explain parts of code to human audiences. In Python, comments start with a `#` symbol. The interpreter ignores everything from the `#` to the end of the line when translating a program into machine instructions.

In [22]:
```
# This is a comment. It is not evaluated when we run our code
```

In [23]:
```
student1_grade = 90
student2_grade = 50
student3_grade = 74
average_grade = (student1_grade + student2_grade + student3_grade) / 3
average_grade  # These lines of code will produce a value.
# Python will ignore everything after the # symbol.
```

Out[23]:

```
71.33333333333333
```

Readability

What is readable code and why should we care?

Code that is easier to read is easier to debug and maintain. Readability becomes more important the larger the codebase or the team is.

Some practices that make code easier to read:

- **White space**: A space to either side of an operand helps readability. We can also split code across multiple lines by wrapping it in parentheses.
- **Comments**: Comments communicate the gist of a program and help explain complicated sections of code. They can also be used to plan future work in plain language, or to toggle lines of code on and off for debugging.
- **Clear and consistent variable names**: Descriptive variable names contribute to self-documenting code and reduce the need for comments. Consistent naming practices reduce the likelihood of referencing the wrong variable or raising a NameError.

Which code is easier to read and follow?

1.
```
studentone_grade=90
StudentGrade2=50
stu3gr=74
avggrade=(studentone_grade+StudentGrade2+stut3gr)/3
```

2.
```
student1_grade = 90
student2_grade = 50
student3_grade = 74
average_grade = ((student1_grade
               + student2_grade
               + student3_grade)
              / 3)
```

Functions

What is a function?

A *function* is a block of code that performs a task. A function takes zero or more inputs and *returns* an output.
When we use a function, we say we are *calling* it. We call a function by typing out the function's name followed by parentheses. Inside the parentheses, we can *pass* in data for the function to use. These values are called *arguments*.

Built-in functions

Python comes with several functions already built in. To display information, for example, we can call the `print()` function. (Note that if the last line in a notebook cell is an expression, the notebook will automatically display the result, no `print()` function needed.)

```
In [24]: print(42)
```

```
42
```

To check the data type of a value, we can call the `type()` function.

```
In [25]: type(42)
```

```
Out[25]:
 int
```

We can pass in expressions as arguments.

```
In [26]: print(10 ** 2)
```

```
100
```

More built-in functions

Some other useful built-in functions include `abs()` for absolute values, `round()` for rounding, and `int()` and `float()` for type conversion into integers and floats, respectively.

In [27]:
```
abs(-43)
```

Out[27]:

43

In [28]:
```
round(2/3)
```

Out[28]:

1

In [29]:
```
int(3.99)
```

Out[29]:

3

In [30]:
```
float(7)
```

Out[30]:

7.0

Nesting functions

We can also *nest* functions within one another. Nested functions are evaluated from the inside out. The code below is evaluated in the following order:

1. `2 * 1.5`
2. `round()`
3. `type()`
4. `print()`

```
In [31]: print(type(round(2 * 1.5)))
```

```
<class 'int'>
```

Getting help

How do we know what a function does or what arguments it can take? One way is by looking up documentation online. Another is by using the built in `help()` function, with the name of the function we want to know about passed in as an argument.

```
In [32]: help(round)
```

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

round(number, ndigits=None)
    Round a number to a given precision in decimal digits.

    The return value is an integer if ndigits is omitted or None.  Otherwise
    the return value has the same type as the number.  ndigits may be negative.
```

Writing our own functions

We can also create our own functions, which allows us to reuse code without copying and pasting. As we write longer programs, functions also help structure our code.

We use the `def` keyword to start a *function definition statement*. The function name follows `def`. Function names can use letters, digits, and underscores; they cannot start with a digit; and they are case sensitive. Functions in Python typically start with lowercase letters.

After the function name come parameter names enclosed in parentheses. A *parameter* is a variable used only within a function. When we pass an argument to a function, the argument value is assigned to a corresponding parameter.

After the closing parenthesis comes a colon (`:`). The function *body* follows on the next lines. The function body is an indented block of code that runs whenever we call the function. It can include comments as well as statements. The body does not run when we define the function, though!

Finally, the body ends with a `return` statement. The return statement tells the function what value, if any, to output when it is called. Some functions, like `print()`, return `None` -- a special data type representing no data. If we don't write a `return` statement, Python will fill in an implicit one and return `None`.

```
degrees_fahrenheit = (9 / 5) * degrees_celsius + 32
```

Let's turn our Celsius-to-Fahrenheit conversion code from earlier into a function.

We want our output to be a number representing degrees Fahrenheit. Our input is a number representing degrees Celsius.
To make it easy to understand what our function does, let's call it `c_to_f()`. We'll need a parameter name for our input within the parentheses as well -- let's use `degrees_c`. Before writing any code in the body, we can add a comment briefly explaining what the function does.
The output is the result of the expression `(9 / 5) * degrees_c + 32`. This calculation is pretty short, so we can put it in the `return` statement as is.

```
In [33]: def c_to_f(degrees_c):
             # Convert degrees from Celsius to Fahrenheit
             return ((9 / 5) * degrees_c + 32)
```

Let's try `c_to_f()` out!

In [34]:
```
print(c_to_f(100))
print(c_to_f(-11))
```

```
212.0
12.2
```

We can save the output of a function by assigning it to a variable.

In [35]:
```
freezing_f = c_to_f(0)
freezing_f
```

Out[35]:

```
32.0
```

Multiple parameters

Functions can have multiple parameters. In this case, the order that arguments are passed in matters. The first argument corresponds to the first parameter, the second argument to the second parameter, and so on.

In [36]:
```
def divide(dividend, divisor):
    return dividend / divisor

print(divide(0, 2))
```

```
0.0
```

In [37]:
```
print(divide(2, 0))
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Input In [37], in <cell line: 1>()
----> 1 print(divide(2, 0))

Input In [36], in divide(dividend, divisor)
      1 def divide(dividend, divisor):
----> 2     return dividend / divisor

ZeroDivisionError: division by zero
```

Optional parameters and default arguments

Functions can have default values for parameters. This is convenient when there is a commonly used default -- it means we do not have to supply that argument if the default value works for our purposes. To override the default, we can pass in an additional argument. Below, we define a sales tax calculator that uses Ontario's tax rate as the default.

In [38]:
```
def calc_sales_tax(price, tax_rate=0.13):
    return price * tax_rate

print(calc_sales_tax(5))
print(calc_sales_tax(5, .08))
```

```
0.65
0.4
```

Keyword arguments

If there are multiple optional parameters, use *keyword arguments* to specify which parameter a value should be used for. Using a keyword argument resembles assigning a variable.
The function below has multiple optional parameters.

In [39]:
```
def calc_total_bill(price, tax_rate=0.13, tip_rate=0.2):
            tax = price * tax_rate
    tip = price * tip_rate
    return price + tax + tip
```

Accept all defaults:

```
In [40]: calc_total_bill(100)
```

```
Out[40]:
133.0
```

Python assumes the second argument is for the second parameter, `tax_rate`.

```
In [41]: print(calc_total_bill(100, 0.22))
```

```
142.0
```

Use the `tip_rate` keyword argument to specify that the second argument is not the tax rate.

```
In [42]: print(calc_total_bill(100, tip_rate=0.22))
```

```
135.0
```

Documenting functions

We can call `help()` on user-defined functions as well as built-in functions.

```
In [43]: help(c_to_f)
```

```
Help on function c_to_f in module __main__:

c_to_f(degrees_c)
```

Docstrings

In order to get useful results, though, a user-defined function needs a docstring. A *docstring* is a special kind of string, or text data, that describes what a function does. It is the first thing in a function after the definition statement, and it is typically surrounded by triple single or double quotes (`'''` or `"""`). `help()` looks for a docstring when it is called.

Let's convert our comment from earlier into a docstring, then try calling `help()` again.

```
In [44]: def c_to_f(degrees_c):
             '''Convert degrees from Celsius to Fahrenheit'''
         return ((9 / 5) * degrees_c + 32)

help(c_to_f)
```

```
Help on function c_to_f in module __main__:

c_to_f(degrees_c)
    Convert degrees from Celsius to Fahrenheit
```

Structure of a function

Now that we've written our first function, let's review the structure.

`c_to_f(degrees_c)` is the *function header*. It tells us what the function name and parameters are.

`'''Convert degrees from Celsius to Fahrenheit'''` is the *docstring*. It briefly describes what the function does. Docstrings can include more information about the function, such as examples, what types of data are accepted as arguments, or what gets returned.

`return ((9 / 5) * degrees_c + 32)` makes up the function *body*, or the code that runs when a function is called. This is typically more than one line of code. The body ends with a `return` statement specifying what the output of the function should be. If no `return` statement is written out, the function will return `None`.

Variable Scope

We mentioned that parameters were variables used only in functions. This means that the parameter does not exist outside of the function call -- if we try to access `degrees_c` outside of `c_to_f()`, we get a NameError.

```
In [45]: degrees_c
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [45], in <cell line: 1>()
----> 1 degrees_c

NameError: name 'degrees_c' is not defined
```

Parameters are *locally scoped* variables. *Scope* refers to the context in which a variable can be accessed. *Local* indicates that the variable can only be accessed within the function itself.

In contrast, a variable defined outside of a function has *global scope*. It can be accessed from anywhere in the program.

In [46]:
```
# boiling_c is a global variable
boiling_c = 100

# boiling_f is too
boiling_f = c_to_f(100)

print(boiling_c, boiling_f)
```

```
100 212.0
```

Function Design Recipe

This function design recipe standardizes the steps to write a function.

1. Think about what your function does. Come up with a few examples of inputs and outputs.
2. Think of a meaningful function name.
3. Write thorough function documentation.
4. Code the function body.
5. Test the function against the examples from step 1.

Practice

Writing functions

Following the function design recipe, define:

- a function that converts Fahrenheit into Celsius
- a function that converts kilometers into miles. (There are 1.6 kilometers in a mile.)

Understanding scope

What value is printed in the below code?

```
answer = 3

def answer_to_everything():
    answer = 42
    return answer

answer_to_everything()

print(answer)
```

Strings

What is a string?

`str`, short for *string*, is Python's text data type. Strings are sequences of characters, including digits and symbols. Strings are surrounded by either single (') or double (") quotes. Which one to use is a matter of preference, but they must be the consistent around a string and should be consistent across our code. The presence of apostrophes, single quotes, or double quotes in our text may require us to use a different quote option, though.

String examples

```
In [47]: 'This is a string'
```

```
Out[47]:
'This is a string'
```

```
In [48]: "So is this"
```

```
Out[48]:
'So is this'
```

```
In [49]: "these quotes do not match'
```

```
  Input In [49]
    "these quotes do not match'
    ^
SyntaxError: unterminated string literal (detected at line 1)
```

In [50]:
```
'Let's see if this works
```

```
  Input In [50]
    'Let's see if this works
         ^
SyntaxError: invalid syntax
```

In [51]:
```
# use double quotes to avoid the error
"Let's see if this works now"
```

Out[51]:

```
"Let's see if this works now"
```

Multiline strings

If we want a string to span multiple lines of code, we need to wrap it in triple quotations (`'''` or `"""`). We saw this before with our docstrings!

In [52]:
```
'''
        I am a multiline string

I'm very useful for documenting functions
and for storing longer texts.

And you don't have to worry about apostrophes or "quote"s!
'''
```

Out[52]:

```
'\nI am a multiline string\n\nI\'m very useful for documenting functions \nand for storing longer
texts.\n\nAnd you don\'t have to worry about apostrophes or "quote"s!\n'
```

Escape sequences

Where did those `\n` characters come from and where did our line breaks go in that last string? `\n` is an *escape sequence*, a combination of characters that means something else. Here, it means "new line", not literally "\n". More generally, the backslash (`\`) is an escape character that can be used to indicate that the next character should be treated differently.

Escape sequences

Escape sequence	Description
\'	Single quote
\"	Double quote
\\|Backslash	
\t	Tab
\n	Newline
\r	Carriage return

```
In [53]: 'This string won\'t result in an error thanks to the escape sequence'
```

```
Out[53]:
"This string won't result in an error thanks to the escape sequence"
```

Working with strings and overloading

Some arithmetic and comparison operators also work with strings, though their meaning changes. When an operator does different things depending on the data provided, we say that it is *overloaded*. It's important to understand what type of data is being used with an overloaded operator so that the results are as expected.

Strings and arithmetic operators

In [54]:
```
# adding strings together concatenates them
'hello' + ' ' + 'world'
```

Out[54]:

```
'hello world'
```

In [55]:
```
# multiplying a string repeats it
'ha' * 3
```

Out[55]:

```
'hahaha'
```

In [56]:
```
# mixing data types results in an error
'The year is ' + 2020
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [56], in <cell line: 2>()
      1 # mixing data types results in an error
----> 2 'The year is ' + 2020

TypeError: can only concatenate str (not "int") to str
```

Practice

Complete the examples in the docstring below, then write the body of the function

In [57]:
```
def repeat(string, num):
        """ Return string repeated num times.
    >>> repeat('yes', 4)
    'yesyesyesyes'
    >>> repeat('no', 0)

    >>> repeat('yesnomaybe', 3)

    """
```

Strings and comparison operators

We can use the `==` and `!=` operators to compare strings, as well as to compare strings and numbers.

```
In [58]: 'apple' == 'Apple'
```

```
Out[58]:
```

```
False
```

```
In [59]: 'apple' != 'Apple'
```

```
Out[59]:
```

```
True
```

```
In [60]: '20' == 20
```

```
Out[60]:
```

```
False
```

Strings within strings

We can extract a piece of a string by *slicing* it. For example, if we want to get initials from a first name and last name, we may slice the first letter of each.
To slice a string, we add square brackets (`[ ]`) to the end of it. To get a single character, we then put the *index* or position, of the character we want to get.
To get multiple characters, we put the starting index of our slice, then a colon (`:`) and finally the index where we want to end our slice. **Strings in Python are *zero-indexed*, meaning that the first letter is at index 0, not 1.** Slices do not include the character at the ending index position.

Slicing single characters

In [61]:
```
first_name = 'Ada'
last_name = 'Lovelace'

# print initials
print('Initials are', first_name[0], last_name[0])
```

```
Initials are A L
```

If we don't provide a starting index, our string slice will go from the beginning to the ending index. Similarly, if we don't provide an ending index, our string slice will go from the starting index to the end.

In [62]:
```
print(first_name[:1], last_name[4:])
```

```
A lace
```

In [63]:
```
phone_number = '+1 555-123-4567'

# get the area code
phone_number[3:6]
```

Out[63]:

```
'555'
```

We can even use negative indices. Here, we slice from the fourth-to-last character to the end.

```
In [64]: phone_number[-4:]
```

```
Out[64]:
 '4567'
```

Checking for strings

We can also check for character sequences, or *substrings* within a string. There are multiple ways to do this, but one of the simplest is with the `in` operator.

In [65]:
```
job_qualifications = 'The successful applicant will be proficient in R, Python, SQL, statistics, and data vis
```

In [66]:
```
'R' in job_qualifications
```

Out[66]:

True

In [67]:
```
' r ' in job_qualifications
```

Out[67]:

False

In [68]:
```
'JavaScript' in job_qualifications
```

Out[68]:

False

Converting between types

Sometimes, we'll get string data that should be treated as a number, or vice versa. We can use the `str()`, `int()` and `float()` functions to convert between data types. If the argument to the type conversion function is not valid in the target data type, Python will produce an error.

STRING TO INTEGER

```
In [69]: int('17')
```

Out[69]:

17

```
In [70]: int('17.4')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [70], in <cell line: 1>()
----> 1 int('17.4')

ValueError: invalid literal for int() with base 10: '17.4'
```

```
In [71]: int('me')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [71], in <cell line: 1>()
----> 1 int('me')

ValueError: invalid literal for int() with base 10: 'me'
```

STRING TO FLOAT

```
In [72]: float('892')
```

```
Out[72]:
892.0
```

```
In [73]: float('you')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [73], in <cell line: 1>()
----> 1 float('you')

ValueError: could not convert string to float: 'you'
```

Float and int to string

```
In [74]: str(37.5)
```

Out[74]:

```
'37.5'
```

```
In [75]: str(20)
```

Out[75]:

```
'20'
```

```
In [76]: '20' + str(37.5)
```

Out[76]:

```
'2037.5'
```

Strings and functions

String data works with some of the functions we've seen before, like `print()`, as well as with built-in functions like `len()`, which tells us how long the argument we pass in is.

In [77]:
```
# We can print strings, ints, and floats together
print(1, 'fish,', 2, 'fish')
```

```
1 fish, 2 fish
```

In [78]:
```
len('Sometimes we type long-winded sentences to meet word counts or character counts.')
```

Out[78]:

```
80
```

String methods

Strings in Python also have *methods*, which are functions that work only for a specific type of data. We'll learn about methods in more depth later. The key points here are:

- String methods work only on strings -- not on integers, floats, or booleans
- Methods are called differently from other functions

To use a string method, we specify a string, then follow it with a period and the method being called.

In [79]:
```
# convert a string to all caps
'I am not yelling'.upper()
```

Out[79]:

```
'I AM NOT YELLING'
```

In [80]:
```
# count the 'e's
'This string is unusual'.count('e')
```

Out[80]:

```
0
```

In [88]:
```
# check if a string ends with a substring
'file_name.csv'.endswith('.csv')
```

Out[88]:

```
True
```

In [89]:
```
# replace text
'long file name with spaces.csv'.replace(' ', '_')
```

Out[89]:

```
'long_file_name_with_spaces.csv'
```

For a full list of string methods, see the **Python documentation**.

Formatting Strings

The `format()` string method makes it easier to combine text with other data. We can create text templates with curly braces (`{}`), then pass values in.

In [81]:
```
'Ada Lovelace\'s initials are {}. {}.'.format('Ada'[0], 'Lovelace'[0])
```

Out[81]:

```
"Ada Lovelace's initials are A. L."
```

If we preface the string with `f`, we can pass values directly into the curly braces.

In [87]:
```
f'Ada Lovelace\'s initials are {"Ada"[0]}. {"Lovelace"[0]}.'
```

Out[87]:

```
"Ada Lovelace's initials are A. L."
```

When in doubt, we can call for `help()` !

```
In [83]: help(str.lower)
```

```
Help on method_descriptor:

lower(self, /)
    Return a copy of the string converted to lowercase.
```

Input

Getting user input

Sometimes, we may want to work with data given to us by the user running the program. One way to get user input is with the `input()` function, which prompts the user and returns their response as a string. `input()` takes a prompt string as an argument.
Be careful when using `input()` -- there's no guarantee that the user will answer in a way you expect!

In [84]:
```
age = input('How old are you? ')
age
```

```
How old are you? 200
```

Out[84]:

```
'200'
```

In [85]:
```
# to do arithmetic on an input, we need to convert it
age_next_bday = int(age) + 1
print('Next birthday you will be {}'.format(age_next_bday))
```

```
Next birthday you will be 201
```

References

- Bostroem, Bekolay, and Staneva (eds): "Software Carpentry: Programming with Python" Version 2016.06, June 2016, **https://github.com/swcarpentry/python-novice-inflammation**, 10.5281/zenodo.57492.
- Chapters 1, 2, 3, and 4, Gries, Campbell, and Montojo, 2017, *Practical Programming: An Introduction to Computer Science Using Python 3.6*
- Chapter 8, Adhikari, DeNero, and Wagner, 2021, *Computational and Inferential Thinking: The Foundations of Data Science*
- "String methods", Python Software Foundation, *Python Language Reference, version 3*. Available at **https://docs.python.org/3/library/stdtypes.html#string-methods**