# Tour of Python

# Tour of Python

In this lesson we'll create two versions of a program that prints a table of corresponding Fahrenheit and Celsius temperatures. Along the way we'll introduce

- values and variables,
- control structures,
- functions, and
- Python scripts.

Experienced programmers will be ready to begin writing Python programs after this lesson.

Astute readers who know C will recognize this example program from the first chapter of Kernighan and Ritchie's classic *The C Programming Language*.

Type the following code into your text editor, save it as `fahrenheit_celsius_v1.py` and run it:

```
lower = 0
upper = 300
step = 20

print(f"Fahrenheit Celsius")
print(f"---------- -------")
fahr = lower
while fahr <= upper:
    celsius = 5 * (fahr - 32) / 9
    print(f"{fahr:<10} {celsius:>7.1f}")
    fahr = fahr + step
```

# Values and Variables

In the *assignment statement*:

```
1 lower = 0
```

- `0` is an `int` literal, that is, the textual representation of an `int` value in Python source code. An `int` literal is a number without a decimal point. A number with a decimal point is a `float` literal.
- `lower` is a variable which, after the assignment statement, reference an `int` object whose value is `0`.
  - Values are strongly typed – Python will not allow type inconsistencies.
  - Variables are dynamically typed – variables don't come into existence until they are assigned values and can be reassigned to values of other types; there is no variable declaration in Python.
  - The types of expressions are not checked until run-time.

# Built-in Functions

`print` is a built-in function in Python. Functions are `Callable` – they are called by placing parentheses after their names. In:

```
1 print("Fahrenheit Celsius")
```

- The `str` literal `"Fahrenheit Celsius"` is the single *argument* to this call to the `print` function.
- `str` values can be enclosed in single or double quotes.
- The `print` function appends a newline character (`\n`) after its argument(s) by default.
  - `print("Fahrenheit Celsius", end='')` leaves off the ending newline.

# `while` Loops

`while` is a loop control structure which has the form:

```
while <continuation_condition>:
    <block>
```

- `while <continuation_condition>:` is the header and must end with a colon, `:`.
- `<block>`, also called a *suite* in Python, is executed repeatedly while the `<continuation_condition>` is "truthy" (a Pythonic word for a value that is treated like `True` in a `bool`ean context).
- `<block>` may be a single statement or sequence of statements and expressions, and must be indented one level beyond the header, typically 4 spaces by the Python style guide, PEP 8.
  - Python does not use braces or begin-end markers for blocks. Indentation has semantic meaning in Python source code and must be consistent – consistent in indentation amount and in the characters used for indentation; you cannot mix TABs and spaces for indentation in the same source file.

# Active Review – `while` Loops

- Identify the components of the `while` loop:

```
1 while fahr <= upper:
2     celsius = 5 * (fahr - 32) / 9
3     print(f"{fahr:<10} {celsius:>7.1f}")
4     fahr = fahr + step
```

- Why is the loop continuation condition guaranteed to become false over successive executions of the loop body?
- What happens if we change the continuation condition to `fahr < upper`?

# f-Strings

```
1 while fahr <= upper:
2     celsius = 5 * (fahr - 32) / 9
3     print(f"{fahr:<10} {celsius:>7.1f}")
4     fahr = fahr + step
```

- `5 * (fahr - 32)/ 9` is an arithmetic expression that produces a `float` value due to the `float` division operator, `/`, so `celsius` references a `float` value.
- `f"{fahr:<10} {celsius:>7.1f}"` is an f-string, short for formatted string literal. The values of expressions enclosed within curly braces are inserted into the string.
  - `{fahr:<10}` means insert a string containing the value of `fahr`, left-aligned within a 10-character field.
  - `{celsius:>7.1f}` means insert a strings containing the value of `celsius`, right-aligned in a 7-character field, formatted as a floating-point value with one digit after the decimal point.

## Active Review

- Experiment with the formatting of the output, e.g., different field widths, alignments, floating-point precision.
  - You can learn more about f-strings in the Python documentation on formatted string literals and the format specification mini-language.

fahrenheit_celsius_v2.py

Parts of fahrenheit_celsius_v1.py are not idiomatic, or "Pythonic". Create a new version, fahrenheit_celsius_v2.py:

```
1  import sys
2
3  def fahrenheit2celsius(f: int) -> float:
4      return 5 * (f - 32) / 9
5
6  def main(args: list[str]) -> None:
7      # Set defaults if no args given
8      if len(args) > 1:
9          lower = int(args[1])
10     else:
11         lower = 0
12     upper = int(args[2]) if len(args) > 2 else 300
13     step = int(args[3]) if len(args) > 3 else 20
14
15     print(f"Fahrenheit Celsius")
16     print(f"---------- -------")
17     for f in range(lower, upper, step):
18         c = fahrenheit2celsius(f)
19         print(f"{f:<10} {c:>7.1f}")
20
21 if __name__=='__main__':
22     main(sys.argv)
```

# Program Structure

```
import sys

def fahrenheit2celsius(f: int) -> float:
    return 5 * (f - 32) / 9

def main(args: list[str]) -> None:
    # Set defaults if no args given
    if len(args) > 1:
        lower = int(args[1])
    else:
        lower = 0
    upper= int(args[2]) if len(args) > 2 else 300
    step = int(args[3]) if len(args) > 3 else 20

    print(f"Fahrenheit Celsius")
    print(f"---------- -------")
    for f in range(lower, upper, step):
        c = fahrenheit2celsius(f)
        print(f"{f:<10} {c:>7.1f}")

if __name__=='__main__':
    main(sys.argv)
```

← Imports appear at the top of the file by convention.

← Functions and classes are next. Definitions must appear before their uses, so the starting point of a Python script is usually at the bottom of the file.

← A distinguished "main" function is optional, but recomended. We'll learn why later.

# is comment character. Everything after # on a line is ignored by Python.

← The "if __name__ - '__main__'" block is the entry point of the program. We'll learn the details when we learn about modules and programs.

# Imports and `if __name__=='__main__'`

To use members of the `sys` module, we must first import it:

```
1 import sys
```

We then use `sys.argv` in the `if __name__=='__main__'` block.

```
1 if __name__=='__main__':
2     main(sys.argv)
```

Every `.py` file whose base name is a legal Python identifier is a Python module. As we'll learn in the lesson on modules and programs, the `if __name__=='__main__'` block is the starting point of a script, and is ignored when a module is imported.

# Functions and Type Annotations

In the function header:

```
1 def main(args: list[str]) -> None:
```

- `def` is a keyword marking a function definition.
- `main` is the name of the function.
- `args` is the name of the single function parameter.
- `: list[str]` is a type annotation that conveys to the programmer that `args` should be a `list` of `str`s. It is ignored by the Python interpreter.
- `-> None` means that `main` returns `None` when called.

Using built-in generic type annotations such as `list[str]` is new in Python 3.9. You'll still see code (possibly in this course!) that uses the older `List[str]` from the `typing` module.

# Function Calls

When `main` is called:

```
1 if __name__=='__main__':
2     main(sys.argv)
```

- Control is transferred to the first statement inside the `main` function.
- `args` becomes an alias for `sys.argv` in the `main` function. Formally, `sys.argv` is an *argument* or *actual parameter* and `args` is a *parameter*. In Python people often use *argument* to refer to both.

# Command-Line Arguments

`sys.argv` is a `list[str]` containing the command-line arguments to the `python3` **program**. In the script invocation:

```
1 python3 fahrenheit_celsius_v2.py 30 100 10
```

`sys.argv` has the value `['fahrenheit_celsius_v2.py', '30', '100', '10']`

Note that all the elements of `sys.argv` are `str`s. If we want to treat any of them as a different data type, we must convert them, as we do in:

```
1         lower = int(args[1])
```

The `int()` constructor parses the `str` contained in `args[1]` and, if it's a valid textual representation of an `int`, returns the `int` value.

## Active Review

- Run your `fahrenheit_celsius_v2.py` script from your OS command-line shell with different values for `lower`, `upper`, and `step`.
  - You will need to run it from the OS shell so that you can provide command-line arguments. You can use your terminal, or you can open an OS shell within PyCharm with OPT-F12 on macOS, or ALT-F12 on Linux or Windows.
  - Can you provide command-line arguments for some of the parameters of the script but not others? Which combinations of arguments can you provide on the command-line?

# `if-else` Statements

In the `if-else` statement:

```
1     if len(args) > 1:
2         lower = int(args[1])
3     else:
4         lower = 0
```

- The `len(args)` function returns the length of the `args` list. Note that `sys.argv` always has the name of the Ptyhon script file as its first element, `sys.argv[0]`, so its length is always $\geq 1$.
- `len(args)> 1` has the value `True` if at least one command-line argument was given.
- If `len(args)> 1` is `True`, the `if` suite is executed, otherwise the `else` suite is executed.

# `if-else` Expressions

`if-else` expressions have the form:

```
<value1> if <condition> else <value2>
```

It has the value <value> if <condition> is truthy, and <value2> if <condition> is falsey.

The expression:

```
    upper = int(args[2]) if len(args) > 2 else 300
```

is an idiomatic way to give `upper` a default value if one is not provided on the command line.

# For Statements and `range` Objects

```
1 for f in range(lower, upper, step):
2     c = fahrenheit2celsius(f)
3     print(f"{f:<10} {c:>7.1f}")
```

- A `range` object is an iterator that produces successive `int`s from `lower` to `upper`, not including `upper`, in increments of `step`.
  - E.g., `range(0, 10, 2)` would produce 0, 2, 4, 6, 8.
- A `for` statement produces a loop in which the loop variable, `f` in this example, assumes the values produced by the iterator after `in` in successive executions of the `for` statement body.

## Active Review

- Run your updated `fahrenheit_celsius_v2.py`. What is the last Fahrenheit value in the table?
- Is our updated version a faithful refactoring (redesign which preserves the behavior of the original program) of the original program?
- How could we modify the new version to match the behavior of the original version?

# Conclusion

When learning or using any language, you need to be familiar with two things: the language specification, and the standard library. As you learn and use Python, keep these links close at hand:

- docs.python.org
- The Python Language Reference
  - You may find The Python Tutorial a more pleasant coverage of the language.
- The Python Standard Library

In the remaining lessons in this course we'll take a deeper dive into all the things we learned in this lesson and more.