Going from Python to C++

C++ intro

Mike Burrell

February 27, 2024

Readings for this set of slides

Chapter 1 —1.1, 1.3, 1.4

Chapter 2 —2.1, 2.2, 2.3, 2.4

Learning objectives

- Refresh existing knowledge from Python course
- Transfer Python knowledge to C++
- Become accustomed to basic C++ syntax

Python hello world


```
1 print('Hello world!')
```

- In Python, this can be a complete program
- The program starts executed from the top line down

C++ hello world


```
1 #include <iostream>
2 int main() {
3     std::cout << "Hello world!" << std::endl;
4     return 0;
5 }
```

- In C++, the program starts executing at the `main` function
- `main` *must* return an `int` (0 if there were no errors)

C++ hello world


```
1 #include <iostream>
2 int main() {
3     std::cout << "Hello world!" << std::endl;
4     return 0;
5 }
```

- In Python, we don't need any `import` or `include` statements for simple (built-in) functions like `print`
- C++ does not have *any* built-in functions or objects
- Printing something out (with `cout`) requires is to include `iostream`

C++ hello world


```
1  #include <iostream>
2  int main() {
3      std::cout << "Hello world!" << std::endl;
4      return 0;
5  }
```

- In C++, all standard functions/objects are part of the `std` *namespace*
- For now, we will prefix them with `std::`

Differences in syntax


```
1  def num_real_roots(a, b, c):
2      discriminant = b ** 2 - 4 * a * c
3      if discriminant < 0:
4          return 0
5      elif discriminant == 0:
6          return 1
7      else:
8          return 2
```

- Here is a Python function which could be used to help calculate the Quadratic Formula ($x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$)

Differences in syntax


```
1  def num_real_roots(a, b, c):
2      discriminant = b ** 2 - 4 * a * c
3      if discriminant < 0:
4          return 0
5      elif discriminant == 0:
6          return 1
7      else:
8          return 2
```

- In Python, *indentation* is extremely important in conveying structure
- Statements are terminated by the *end-of-line*

Differences in syntax


```
1  int num_real_roots(double a, double b, double c) {
2      double discriminant = b * b - 4 * a * c;
3      if (discriminant < 0) {
4          return 0;
5      } else if (discriminant == 0) {
6          return 1;
7      } else {
8          return 2;
9      }
10 }
```

- Here is the equivalent function in C++

Differences in syntax


```
1  int num_real_roots(double a, double b, double c) {
2      double discriminant = b * b - 4 * a * c;
3      if (discriminant < 0) {
4          return 0;
5      } else if (discriminant == 0) {
6          return 1;
7      } else {
8          return 2;
9      }
10 }
```

- The C++ compiler completely ignores indentation
- We still indent, for the sake of humans reading it

Differences in syntax


```
1  int num_real_roots(double a, double b, double c) {
2      double discriminant = b * b - 4 * a * c;
3      if (discriminant < 0) {
4          return 0;
5      } else if (discriminant == 0) {
6          return 1;
7      } else {
8          return 2;
9      }
10 }
```

- Structure in C++ is indicated by { } curly braces
- Statements are terminated by ; semicolons

Differences in syntax


```
1  int num_real_roots(double a, double b, double c) {
2      double discriminant = b * b - 4 * a * c;
3      if (discriminant < 0)
4          return 0;
5      else if (discriminant == 0)
6          return 1;
7      else
8          return 2;
9  }
```

- If there is only 1 statement in a block (e.g., after an `if`), the { } curly braces are optional

Differences in syntax


```
1  int num_real_roots(double a, double b, double c) {
2      double discriminant = b * b - 4 * a * c;
3      if (discriminant < 0)     return 0;
4      else if (discriminant == 0) return 1;
5      else                      return 2;
6  }
```

- Putting each statement on a separate line is also optional

Type annotations


```
1  def num_real_roots(a, b, c):
2      discriminant = b ** 2 - 4 * a * c
3      if discriminant < 0:
4          return 0
5      elif discriminant == 0:
6          return 1
7      else:
8          return 2
```

- Python is a *dynamically-typed* language
- The programmer (you) does not explicitly put any type information in the source code

Type annotations


```
1  int num_real_roots(double a, double b, double c) {
2      double discriminant = b * b - 4 * a * c;
3      if (discriminant < 0) {
4          return 0;
5      } else if (discriminant == 0) {
6          return 1;
7      } else {
8          return 2;
9      }
10 }
```

- C++ is a *statically-typed* language
- Every return value and every variable must be given a type
  - In this case, `int` and `double`

if statements


```
1  if (discriminant < 0) {
2      return 0;
3  } else if (discriminant == 0) {
4      return 1;
5  } else {
6      return 2;
7  }
```

- C++ does not have a special `elif` form
  - It is a combination of `else` and `if`
- Like in Python, `else` is optional
- The conditional must be in `(` `)` parentheses

while loops


```
1  int x = 1;
2  while (x < 100) {
3      std::cout << x << std::endl;
4      x *= 2;
5  }
```

- The `while` loop is quite similar to in Python
- As with the `if`, the condition must be in `( )` parentheses

while loops


```
1 int x = 100;
2 do {
3     std::cout << x << std::endl;
4     x *= 2;
5 } while (x < 100);
```

- Unlike in Python, C++ has a `do-while` variant of the `while` loop
- The only difference is that a `do-while` is guaranteed to execute at least once

for-loops


```
1 for i in range(50):
2     print(i)
```

- Python's `for`-loops require a list or sequence on the righthand side
- C++ has those, too, but we are not advanced enough for them yet
- To start with, we will focus on the common pattern of using `range`

for-loops


```
1 for i in range(50):
2     print(i)
```

- The `range` function in Python creates an iterable sequence that goes from one number to another
- `range(50)` produces the numbers $0, 1, 2, 3, \ldots, 49$, in sequence
- We can accomplish the same thing in C++, but without creating a sequence

for-loops


```
1 for (int i = 0; i < 50; i += 1) {
2     std::cout << i << std::endl;
3 }
```

- The most basic form of a `for`-loop in C++ has 3 components to it
- It is sometimes called a *counting* loop

for-loops


```
1 for (int i = 0; i < 50; i += 1) {
2     std::cout << i << std::endl;
3 }
```

`int i = 0` — the first part is the *initialization*, and is a statement which is executed *before* the loop. It introduces the variable and indicates the value we *start* looping at

`i < 50` — the middle part is the *condition*, and indicates when we should continue looping. It specifies the value we *stop* looping at

`i += 1` — the last part is the *incrementing* step and is executed after *each* iteration of the loop. It specifies *how* the value changes each time

Incrementing and decrementing


```
1 for (int i = 0; i < 50; i += 1)
2     std::cout << i << std::endl;
3 for (int i = 0; i < 50; i++)
4     std::cout << i << std::endl;
```

- These two loops are identical
- Because incrementing or decrementing by 1 is so common in C++, C++ offers a special shortcut syntax: ++ and --

Conclusion

In C++, we have a cursory idea of how to:

- Structure a program with a `main` function
- Print things out to the screen
- Declare variables
- Make functions
- Use logical flow control structures (`if`, `while`, `for`)

Now let's see some examples!

Number of divisors

Class exercise

Let's write a complete C++ program which reads in a positive integer n from the user and prints out, for all integers from to 1 to n, how many divisors that number has. Since we don't know much fancy formatting yet, we'll use tab stops to line up the columns.

Approximation a cubic root

Class exercise

Let's use Newton's Method to approximate a root of a cubic. Newton's Method provides a convering sequence of estimates of the form:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

where x_0 can be chosen arbitrarily.
For example, $f(x) = x^3 - 2x^2 - 11x + 12$ has roots at -3, 1, 4, and we should be find one of those roots (varying on the initial guess for x_0).