Information

Monday, June 3, 2024 9:55 AM

Certificate Structure: 19 Chapters over 4 Courses

Fall Semester starts August 19, 2024 (54 work days); 2.8 days per chapter

Grading:

- Problem sets: 70%
  - 3 total (Chapters 2.2, 2.3, 2.4)
  - Unlimited attempts
- Test: 30%
  - Limited attempts
- Practice tests will be given in preparation for the test, which will be 2 hours timed and include multiple choice, fill in the blank, and live coding problems.

Course Feedback Form HERE.

Textbook HERE.

Links

Monday, June 3, 2024 11:33 AM

Helpful Resources

Foreword to How to Think Like a computer Scientist

What is Code?

Learn Python the Hard Way

Code Academy's Python Course: interactive coding practice

Google's Python Course: free python lessons and exercises

CS50: Harvard University's Intro to Computer course

Python Error Encyclopedia

Debugging Guide

Code Shrew

1.1 Computing

Monday, June 3, 2024 9:59 AM

Programming Vocabulary

Programming: the act of creating instructions (code) for a computer to carry out.

Line of code: a single instruction for the computer to perform.

Program: an independent collection of lines of code that serves one or more functions.

Input: data that is fed to a program for it to operate upon.

Output: what the computer provides in return after running some lines of code.

Compile: to translate human-readable computer code into instructions the computer can execute. In the programming flow, this functions as a check on the code the user has written to make sure it makes sense to the computer.

- Languages that require compilation are called Static.
- Languages that do not require compilation are called Dynamic; Python is dynamic.
- Compiling is like checking your code beforehand to ensure it makes sense and it also translates the code into low-level commands the computer can execute.

Execute: running some code and having it actually perform its operations.

- A program doesn't actually perform its task until it's executed.

Programming Languages

Popularity Index of different programming languages HERE.

Console vs GUI

Graphical User Interfaces (GUIs) are systems that involve any kind of output beyond just text and input beyond text entry; buttons, tabs forms, etc. Web browsers, word processors, smartphone apps, Word/PowerPoint, and modern operating systems are all GUIs.

Consoles are an output medium for a program to show exclusively text-based output.

Computing vs Programming

Programming is like learning how to speak the language of the computer while Computing is about what you use the language to say. Computing concepts apply to many different programming languages.

This course splits the material into three categories:

- Foundations: the core principles of computing that transcend any one language
- Language: computing principles implemented in specific languages
- Domain: applying computing principles in a specific language to a particular domain, such as data science, robotics, or digital media.

Course Outline

The course is divided into five units:

1. Basics
    a. Computing: the basic principles of working with computers
    b. Programming: the general workflow of writing and running programs
    c. Debugging: the processing of finding and fixing errors in programs
2. Procedural Programming
    a. Procedural Programming: the general idea of writing sequences of instructions for the computer to perform.
    b. Variables: creating and modifying data in our programs.
    c. Logical Operators: establishing the truth or falsehood of relationships among variables in our programs.
    d. Mathematical Operators: using arithmetic operators to modify the values of variables in our programs.
3. Control Structures
    a. Control Structures: the general idea of lines of code that can control other lines of code.
    b. Conditionals: lines of code (called if-statements) that check logical expressions to see if certain code blocks should run.
    c. Loops: lines of code that instruct the computer to repeat a block of code until some condition is met.
    d. Functions: miniature programs within a larger program, each with their own input, code, and output.
    e. Exception Handling: lines of code that instruct the computer how to fail gracefully when errors are encountered.
4. Data Structures
    a. Data Structures: the general idea of data types more complex than individual letters and numbers.
    b. Strings: ordered series of characters that often represent natural human language.
    c. Lists: ordered series of other kinds of data, collected under one variable name and accessed via numeric indices.
    d. File Input/Output: writing a program's data to a file so it can later be re-loaded after the program is closed and reopened.
    e. Dictionaries: pairs of keys and values collected under one variable name, like lists with non-numeric indices.
5. Objects and Algorithms
    a. Objects: creating and using custom data types so our programs can reason about the world the way we do.
    b. Algorithms: complex sequences of instructions that transform data or generate useful conclusions.

Python

Note: Python 2 and Python 3 are not compatible with one another; this course works with Python 3.

Python is a high-level dynamic programming language; this means that it extracts pretty far away from the core processor and memory of the computer so we don't have to worry about things like managing memory and the language is more portable. This means Python is easier to use than lower-level languages.

Since Python is dynamic, it will run our code without trying to compile it first line by line. As a result, we may not be fully aware of errors until we actually run the lines of code.

This course will be largely completed using integrated Vocareum directly in the browser.

1.2 Programming

Monday, June 3, 2024 9:59 AM

Programming

Programming: Writing code through an iterative process of writing lines of code, attempting to execute them, and evaluating results. Used interchangeably with "coding".

The first program we'll write is simple, with only one line of code:

```
print("hello world")
```

print(): is a function that takes a string of characters as input and prints it to the console. A function is a command we can call in order to do something semi-complex; functions generally are formatted with parentheses at the end.

Compiling

Compiling is a process in which the computer reads over the code before executing it to check for obvious errors and also translating the code we write into low-level instructions that the computer understands. If there is an error in line 5, the compiling process will prevent any of the lines from running until the error is fixed.

Every programming language is "compiled" in the sense that compilation is necessary to translate the code into instructions the computer can understand. The difference between static and dynamic languages is whether or not all of the code is compiled before any of it is run.

For static languages (like Java and C), the entire program must be compiled before it can run. For dynamic languages (like Python and JavaScript), the lines of code are compiled and run one at a time.

Static is also called compiled, while Dynamic is also called dynamic or interpreted.

Executing

Even if we've successfully compiled the program, it's possible to still encounter errors during the execution / running step. Perhaps the program didn't give the output we expected or an important piece of information was missing, etc. These are either errors, which occur when the code attempts to do something it isn't allowed to do, or incorrect results.

1.3 Debugging

Monday, June 3, 2024 9:59 AM

Debugging

Debugging: resolving problems in code, whether it be errors thrown in compilation or running or mismatches between desired and observed output.

Joyner's Law of Debugging: the time required to fix an error is inversely proportional to the time required to find the error. The longer it takes to find the error, the easier it is to fix once found (usually).

Error Types

There are three high-level results of running code:

1. Perform correctly
2. Perform incorrectly
3. Generate an error
    a. Compilation errors
    b. Runtime errors

Compilation errors occur during the computer's read-through of the code. They are an error inherent within the code, not an error that only exists in the context of how the code is run. Some common types of compilation errors include:

- Syntax errors: code that doesn't work with the current programming language
- Name errors: code that tries to use something that doesn't exist
- Type errors: code that doesn't make sense, such as asking for the color of number 5

Runtime errors occur when trying to actually execute the code. If a language doesn't do compilation, it will only have runtime errors. Runtime errors are generally harder to solve that compilation errors. Some common types of runtime errors include:

- Divide by Zero errors: code that divides a value by zero
- Null errors: code containing a variable that has no value
- Memory errors: code that surpasses the computer's memory

Python Error Types

NameError: an error that usually occurs when you use a variable name that doesn't exist yet.

- Resolution: check first for misspellings of the variable. If that doesn't work, try to find where the variable was first defined.

TypeError: an error that occurs when you try to perform an operation on an object that doesn't make sense with the operation, such as calculating the length of a number or printing an omelet. It occurs when you've tried to perform an operation on something of the wrong type.

AttributeError: an error that occurs when you ask for information about a variable that doesn't make sense, like the happiness of a potato. It occurs when you've tried to access a non-existing attribute for an object.

SyntaxError: an error that occurs when the line of code you've written can't be read by the computer because it doesn't match the computer's expectation for the programming language's grammar. This is a sort of catch-all error and is the trickiest to debug.

Note: If you have an unclosed parenthesis, it will result in a SyntaxError on the following line of code. If the error is on the last line of code, you'll get an error that says "unexpected EOF while parsing" (end of file while reading).

Debugging

The goal of debugging is to get the information necessary to locate and fix the error.

Print Debugging: aka tracing. A form of debugging where print statements are added throughout the code to check how the program is flowing.

```
i = 10
count = 1
print("Starting first loop...")
while count < i:
    count = count + 1
print("First loop done.")
print("Starting second loop...")
while count > 0:
    count = count + 1 # incorrect line, should subtract instead
print("Second loop done.")
print(count)
```

Scope Debugging: a form of debugging where print statements are added to check the status of the variables in the program at different stages to see how they are changing.

```
print("Calculating...")
grades = [100, 95, 93, 91, 90, 89, 87, 87, 85, 85, 84, 82]
sum = 0
count = 0
for grade in grades:
    count = count + 1
    print("Adding grade ", count, "...", sep="") # confirm each grade is added
    sum = grade # this is the incorrect line
    total = grade + sum # corrected
    print("Current sum:", total) # check current total
print(total / count)
```

Rubber Duck Debugging: a form of debugging where the programmer explains the logic, goals, and operations to an inanimate listener to methodically step through the code.

Most Integrated Development Environments (IDEs) supply advanced debugging functionalities:

1. Step-by-Step Execution: runs code one line at a time so you can trace through the code and ensure it is running as expected.
2. Variable Visualization: some programs provide a window where at any time you can view the status of a variable in the code, instead of only when it is explicitly printed.
3. In-Line Debugging: while writing your code, the program will identify where you have gone wrong. This is usually visualized like a spellcheck.

2.1 Procedural Programming

Monday, June 3, 2024 9:59 AM

Programming

Procedural Programming: the most basic type of programming, built around the idea that programs are sequences of instructions to be executed.

Functional Programming: programming that makes heavy use of functions and methods to build programs.
- Function: a segment of code that performs a specific task, sometimes around taking some input and returning an output.
- Method: a function that is part of a class in object-oriented programming; colloquially, often used interchangeably with function.

Object-Oriented Programming: a programming paradigm where programmers define custom data types that have custom methods embedded within them. We create objects and "teach" the computer how to use them.

Event-Driven Programming: a type of programming where the program generally awaits and reacts to events rather than running code linearly. Think like a word processor: when you aren't typing, the program isn't just doing nothing, it's waiting for you to type and then it reacts to that event.

Some fundamental concepts to learn for procedural programming:
- Data types
- Variables
- Logical operators
- Mathematical operators

Comments: notes from the programmer supplied in-line alongside the code itself, designated in such a way that the computer is prevented from reading or executing them as code.

Documentation: collected and set-aside descriptions or instructions for a body of code.

Self-Documenting Code: code whose variables and functions are named in such a way that it makes it clear what their underlying content and operations are intended to do.

Syntax

Different programming languages have different syntaxes when it comes to line-ending behaviors and comments. See the guides below:

Language	Syntax	Notes
Python	`print("Here's a line of code")` `print("Here's another one!")`	By default, `print()` adds a line break to the end of the printed text.
Java	`System.out.println("Here's a line of code");` `System.out.println("Here's another one!");`	Like Python's `print()`, Java's `System.out.println()` adds a line break after the printed text. `System.out.print()` will print some text without creating a new line afterward.
C	`printf("Here's a line of code");` `printf("Here's another one!");`	In C, the `printf()` function does not add a line break after the printed text. To do that, you need to include "\n" inside the printed text itself, e.g. `printf("New line please!\n")`.
C++	`std::cout << "Here's a line of code";` `std::cout << "Here's another one!";`	Like C, C++'s `cout` will not add a new line after the printed text. To add one, we would need to write `std:cout << "New Line please!\n"`.
C#	`Console.WriteLine("Here's a line of code");` `Console.WriteLine("Here's another one!");`	Like Java, C# also has a `Console.Write()` function that will write without starting a new line afterward.
JavaScript	`console.log("Here's a line of code")` `console.log("Here's another one!")`	By default, `console.log()` adds a line break to the end of the printed text.
VB.NET	`Console.WriteLine("Here's a line of code")` `Console.WriteLine("Here's another one!")`	C# and VB.NET use the same underlying library, so you'll find that oftentimes syntax like `Console.WriteLine` is shared between them.
Matlab	`fprintf("Here's a line of code);` `fprintf("Here's another one!");`	Like C, Matlab's `fprintf` will not add a new line after the printed text. To add one, we would need to write `fprintf("New Line please!\n")`.
Swift	`print("Here's a line of code")` `print("Here's another one!")`	By default, `print()` adds a line break to the end of the printed text.
Ruby	`puts "Here's a line of code"` `puts "Here's another one!"`	By default, `puts` adds a line break to the end of the printed text.

Language	Syntax	Notes

Python	`#This is a comment in Python`
	
	`#This is a`
	`#multiline comment`
	`#in Python`
		Python has no special way of creating multi-line comments.

Java	`//This is a comment in Java`
	
	`/*`
	` * This is a`
	` * multi-line comment`
	` * in Java`
	` */`
		The asterisks at the beginning of each line inside the multi-line comment are customary, but not required. Java, C++, JavaScript, and C# use the same syntax.

C	`/* This is a comment */`
	
	`/* This is a`
	` * multiline comment`
	` * comment in C`
	` */`
		The asterisks at the beginning of each line inside the multi-line comment are customary, but not required. C does not have a dedicated single-line comment syntax.

C++	`//This is a comment in C++`
	
	`/*`
	` * This is a`
	` * multi-line comment`
	` * in C++`
	` */`
		The asterisks at the beginning of each line inside the multi-line comment are customary, but not required. Java, C++, JavaScript, and C# use the same syntax.

C#	`//This is a comment in C#`
	
	`/*`
	` * This is a`
	` * multi-line comment`
	` * in C#`
	` */`
		The asterisks at the beginning of each line inside the multi-line comment are customary, but not required. Java, C++, JavaScript, and C# use the same syntax.

JavaScript	`//This is a comment in JavaScript`
	
	`/*`
	` * This is a`
	` * multi-line comment`
	` * in JavaScript`
	` */`
		The asterisks at the beginning of each line inside the multi-line comment are customary, but not required. Java, C++, JavaScript, and C# use the same syntax.

VB.NET	`'This is a comment in VB.NET`
	
	`'This is a`
	`'multiline comment`
	`'in Python`
		VB.NET has no special way of creating multi-line comments.

Matlab	`%This is a comment in Matlab`
	
	`%[`
	`This is a`
	`multiline comment`
	`in Matlab`
	`%]`	

Swift	`///This is a comment in Swift`
	
	`/**`
	` This is a`
	` multi-line comment`
	` in JavaScript`
	`*/`
		Swift differs in subtle ways from C++, C#, Java, and JavaScript: it uses three slashes for a single-line comment, two asterisks at the start of a multi-line comment, and does not by convention put asterisks at the start of lines inside a multi-line comment.

Ruby	`#This is a comment in Ruby`
	
	`=begin`
	`This is a`
	`multiline comment`
	`in Ruby`
	`=end`
		Ruby is weird.

2.2 Variables

Monday, June 3, 2024 10:00 AM

Variables

Variables: alphanumeric identifiers that hold values like integers, strings, and dates. Using the equals sign to set a variable to some value is known as assignment. The variable type will match the type of the value assigned to it.

If you use a variable without first giving it a value, it will generally trigger something called a Null Pointer Exception.

Data Types

Data Type: the type of content a variable holds, such as an integer or string. In strongly-typed languages, assigning a type to a variable is a separate step from assigning the value to a variable, but Python does this step automatically. You can use the function type() to check a variable's type.

- Integers / whole numbers
- Floats / real numbers
- Characters
- Strings / collections of characters
- Booleans

You can also create your own types.

Note: if you try to print two different data types together, combined with a +, you'll get a type error.

None: this is Python's implementation of null, a variable that has no value. It doesn't fit into any of the usual data types listed above, it instead has its own data type called NoneType.

```
num_1 = 17
num_2 = 6

quotient = print("The quotient is", num_1 // num_2)
print(quotient) # this prints None because quotient is the result of a print statement, not an
actual value.
correct_quotient = num_1 // num_2
print("The quotient is", correct_quotient)

remainder = print("The remainder is", num_1 % num_2)
print(remainder) # this prints None because remainder is the result of a print statement, not an
actual value.
correct_remainder = num_1 % num_2
print("The remainder is", correct_remainder)

print("We're done!")
```

Type Conversions

str(variable): take as input some variable and return a string representation of the variable's value. Every data type can be converted to some kind of string.

int(variable): take as input some variable (usually a string) an attempt to convert it to an integer, returning the integer if successful or raising a ValueError otherwise. This function works if the variable is a string made up of only digits and/or the negative sign.

bool(variable): take as input some variable (usually a string) and attempt to convert it to a boolean, raising a ValueError if unsuccessful. Generally, false is 0 and true is anything else.

float(variable): take as input some variable (usually a string) and attempt to convert to a float, raising a ValueError if unsuccessful.

This function will work only if the string is made up of digits, a negative sign, and/or a decimal point.

input (string): take as input a user's entry, returned as a string.

Note: when working with inputs, if you try to convert the string to a type (such as *int(input(string))*) and the string isn't actually convertible to that type, there are two ways to handle it:

- Check first to see if the input is an integer before attempting the conversion
- Try the conversion and tell the computer how to handle it if it hits an error.

Reserved Keywords

When naming variables, you have to be sure you don't use any of Python's reserved words. If you try to use them out of their appropriate context, it confuses the computer. If you misuse a reserved word, a SyntaxError will occur.

Below are lists of all Python's reserved words.

Importing Libraries:

- import
- from

Logical Operators:

- and
- is
- not
- or
- False
- True
- None

Functions:

- def
- return

Control Structures:

- as
- break
- continue
- if
- elif
- else
- for
- in
- while
- pass
- with

Object-Oriented Programming Syntax:

- class

Error Handling:

- except
- finally
- raise
- try

Dot Notation

Dot Notation can be used to see additional information stored within a variable. They can be thought of similarly to apostrophes in

English; *person.name* is like *person's name*.

```
from datetime import date
my_date = date.today()
print(my_date.year)
print(my_date.month)
```

2.3 Logical Operators

Monday, June 3, 2024 10:00 AM

Operators

Operators: specific, simple functions that act on primitive data types. These can commonly split into two categories: mathematical and logical operators.

Mathematical Operators: operators that perform mathematical functions. Typical operators include: addition, subtraction, multiplication, division, and modulus (remainder).

Logical Operators: operators that perform logical values, such as comparing relative values, checking equality, checking set membership, or evaluating combinations of other logical operators. These operators have two outputs: True or False. Logical operators can be further divided into two categories: relational and boolean operators.

- relational operators check if things are true
- boolean operators check the combination of multiple relational operators

Relational Operators

Relational Operators: operators that check relationships between multiple variables.

Numeric Comparison Operators: operators that facilitate numeric comparison between values:

- greater than >
- greater than or equal to >=
- equal to ==
- less than <
- less than or equal to <=

```
a = 5
b = 5
c = 3
print(a == b)
print(b == c)
print(a == c)

print(5 > 5)
print(5 >= 5)
print(5 == 5)
print(5 <= 5)
print(5 < 5)
```

Note: a string will be "less than" another string if it comes first alphabetically. A the same letter capitalized will come before the lowercased letter.

Non-Numeric Comparison: nearly any kind of data can compare for equality, even if it isn't numeric. In practice, this can sometimes mean that the values are compared to see if they are the same, or if they point to the same data in memory, or if they have the same data stored in different places.

```
string1 = "Welcome to "
string2 = "CS1301!"
string3 = "CS1301!"
string4 = "Welcome to CS1301!"
print(string1 == string2)
```

```
print(string2 == string3)
print(string3 == string4)
print(string1 == string2 + string4)
print(string1 + string2 == string4)

print("Apple" < "Apple")
print("Apple" < "Grape")
print("Apple" < "apple")
print("apple" < "Raisin")
#print("Apple" < 5)
print("Quick" < "Quicker")
print("Quickly" < "Quicker")
print("Quickly" < "Quick as the wind")
```

Set Operators: operators that check to see if a value is a member of a set of multiple values, usually stored in a string or list. fruit_list = ['apple', 'orange', 'pear'] contains 'apple', but not 'pomegranate'. The in operator checks to see if something is contained within a list of other things.

Note: since an empty string "" technically has nothing to be found, that means it is found. Therefore the in operator will always return True for an empty string.

```
list_of_fruits = ["grape", "orange", "banana", "kiwi", "pear"]
search_fruit = "kiwi"
print(search_fruit in list_of_fruits)
```

Boolean Operators

Boolean Operators: operators like "and" and "or" that act on pairs of boolean values or that act on single boolean values, such as "not".

And: an operator that acts on two boolean values and evaluates as True if and only if they are both true.

Or: an operator that acts on two boolean values and evaluates as True if at least one of them is true.

Not: an operator that acts on one boolean value and evaluates to the opposite value.

```
hungry = True
coworkers_going = False
brought_lunch = False
#Imagine you're deciding whether or not to go out to lunch.
#You only want to go if you're hungry. If you're hungry, even
#then you only want to go if you didn't bring lunch or if
#your coworkers are going. If your coworkers are going, you
#still want to go to be social even if you brought your lunch.
print(hungry and ((not brought_lunch) or (brought_lunch and coworkers_going)))
```

Truth Tables

Truth tables: tables that map out the results of a statement in boolean logic depending on the values of the individual variables.

A	B	(A and B)	(A or B)	(not B)
True	True	True	True	False
True	False	False	True	True
False	True	False	True	False
False	False	False	False	True

A	B	(A and not B)	(not A or not B)	not (A or B)
True	True	False	False	False
True	False	True	True	False
False	True	False	True	False
False	False	False	True	True

A	B	C	A and (B or C)	(A and B) or (B and C)	C or not (A and not B)
True	True	True	True	True	True
True	True	False	True	True	True
True	False	True	True	False	True
True	False	False	False	False	False
False	True	True	False	True	True
False	True	False	False	False	True
False	False	True	False	False	True
False	False	False	False	False	True

Properties of Boolean Operators

Commutative Property: (A and B) is the same as (B and A)

Distributive Property: (A and (B or C)) is the same as ((A and B) or (A and C))

De Morgan's Law (and → Or): (not (A and B)) is the same as (not A or not B))

Order of Logical Operators

1. Whatever is inside parentheses
2. Not
3. And
4. Or

2.4 Mathematical Operators

Monday, June 3, 2024 10:00 AM

Operators

Assignment Operator: an operator that takes the output of an expression and assigns it to a variable.

Mathematical Operator: an operator that mimics basic mathematical functions, like addition and multiplication.

Modulus: the remainder function, which returns the remainder of one number divided by another.

len(): a funciton that takes as input a variable with a length, such as a string of characters or list of items, and returns its length.

// : Floor Division - division that rounds down to the nearest whole number.

**: Exponentiation - raises the first number to the second number.

Python's seven basic operators:

- addition +
- subtraction -
- multiplication *
- division /
- modulus %
- floor division //
- exponentiation **

Note: division and floor division have equal priority, so evaluate left to right. Division results in a float while floor division results in an integer.

Self Assignment & Incrementing

Self-Assignment: a common programming pattern where a variable is assigned to the output of an expression that includes the variable itself.

```
my_var = 1
my_var = my_var + 5
my_var = my_var - 10
my_var = my_var * 3
my_var = my_var / 6
my_var = my_var // 2
my_var = my_var ** 3
```

Python has a shortcut formatting to increment, with the operator symbol next to an equals sign.

```
my_var = 5
my_var += 7
my_var /= 3
my_var **= 3
```

Increment: repeatedly adding a constant (typically one) to a variable. This is usually accomplished using loops, which is a series of code that runs a set number of times.

```
letterCount = 0
for character in "Hello, world":
    #Add one to letterCount
    letterCount += 1
print(letterCount)
```