

# 00000101

## CHAPTER 5

# Computers, Programs, And Algorithms

Ch-5: Computers, Programs, And Algorithms

### Learning Objectives

- *State the purpose and use of a computer*
- *Explain in your own words what makes a computer a unique device*
- *List and describe the four stages of the program execution cycle*
- *Explain how a computer stores and retrieves programs for execution*
- *State the difference between a computer and a computer system*
- *Define the concept of a program from both the human and computer perspective*
- *State the purpose of main, auxiliary, and cache memory*
- *Describe how programs are loaded into main memory and executed by a computer*
- *State the difference between a compiler vs. an interpreter*
- *State the purpose and use of the Python interpreter*
- *Describe how Python programs are loaded and executed by the Python interpreter*
- *Describe what gets stored in the pycache directory*
- *State the definition of an algorithm*

0  
0  
0  
0  
0  
0  
1  
0  
1

---

## INTRODUCTION

---

Computers, programs, and algorithms are three closely related topics that deserve special attention before you dive deeper into Python. Why? Simply put, computers execute programs and programs implement algorithms. As a programmer you will live your life in the world of computers, programs, and algorithms.

As you progress through your studies you will find it helpful to understand what a computer is, what particular feature makes a computer a truly remarkable device, and how a computer functions from a programmer's point of view. You will also find it helpful to know how humans view programs and how human-readable program instructions are translated into computer-executable form.

Next, it will be imperative for you to thoroughly understand the concept of an algorithm and to understand how good and bad algorithms ultimately affect program performance.

Finally, I will show you how Python programs are transformed into bytecode and executed by the Python runtime.

Armed with a deeper understanding of computers, programs, and algorithms, you'll be better informed and able to write better software.

---

## 1 WHAT IS A COMPUTER?

---

A computer is a device whose function, purpose, and behavior is prescribed, controlled, and changed via a set of stored instructions. A computer can also be described as a general-purpose machine. One minute a computer may function as a word processor or page-layout machine, and the next minute it may function as a digital canvas for an artist. This functionality is implemented as a series of instructions. Indeed, in each case the only difference between the computer functioning as a word processor and the same computer functioning as a digital canvas is in the set of instructions the computer executes. This is what makes a computer a truly remarkable device — it's a changeable machine.

### 1.1 COMPUTER VS. COMPUTER SYSTEM

Due to the ever-shrinking size of the modern computer, it is often difficult to separate the concept of a computer from the computer system in which it resides and it doesn't help that the term computer is used to refer to a complete system, like a laptop, and to the processor that powers it. As a programmer, you will be concerned with both.

You will need to understand issues related to the particular processor that powers a computer system in addition to issues related to the computer system as a whole. Luckily though, you can be extremely productive armed with only a high-level understanding of each. Ultimately, I highly recommend spending the time required to get intimately familiar with how your computer operates. In this chapter I use my personal Apple MacBook Pro<sup>®</sup> 16-inch laptop with an M1 Max processor as an example but the concepts are the same for any computer or computer system.

### 1.1.1 COMPUTER SYSTEM

My Apple MacBook Pro laptop is pictured in figure 5-1.



Figure 5-1: Typical Apple Mac Pro Computer System

Referring to figure 5-1 — An essential element of any respectable laptop is a tricked-out lid. Strangely enough, I also sign my laptops and write “*Signature Model*” under my initials. I then dive into my collection of cool stickers obtained from various things I’ve purchased over the years like guitar pedals (*TC Electronics*) or *Peak Design* camera gear, discontinued code editors (Atom), companies I’ve worked at (*IronNet Cybersecurity*), and various conferences I’ve attended (*AWS re:Invent*). (A big shout-out to my dear friend *Tri Nguyen*, who offered me his most awesome collection of stickers to choose from when I was running low!)

The *computer system* comprises the laptop body with built-in keyboard, trackpad, speakers, display, camera, microphones, and various ports along either side of the laptop body to interface with external peripheral devices. Radio transmitters and receivers are included to connect to wireless networks and Bluetooth devices. The computer system also includes any operating system or utility software required to make all the components work together. This laptop runs the latest version of *macOS*.

The laptop body houses the *system unit* or *system board* and other components. To see inside the laptop you need to remove the back cover. The laptop’s internal components are shown in figure 5-2.

Referring to figure 5-2 — The system unit contains the battery system, which occupies quite a lot of space as you can see. It has a total of 8 terabytes of SSD flash memory storage and a robust

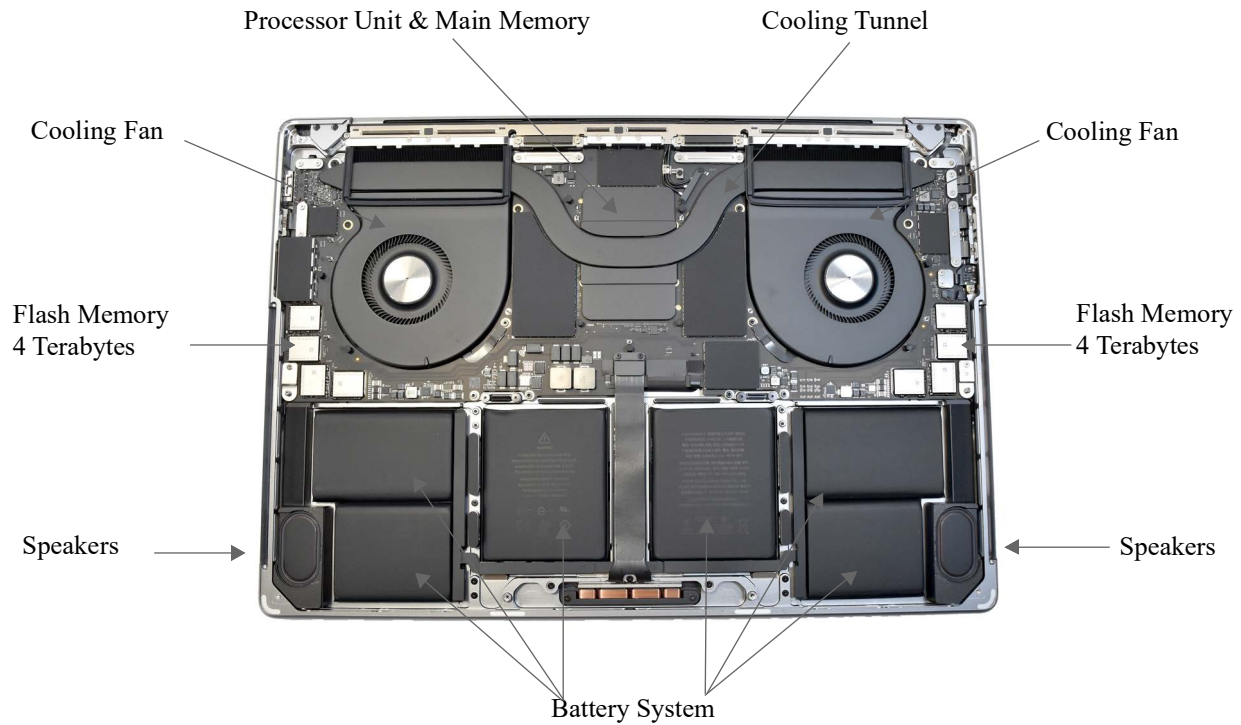


Figure 5-2: Apple MacBook Pro 16-Inch M1 Max Laptop System Internals

cooling system which consists of two turbine-style fans and a cooling tunnel that moves cooled air across the main processor housing. Let's zoom in on the processor unit.

### 1.1.2 PROCESSOR

Figure 5-3 shows a close-up x-ray view of the main processor unit.



Figure 5-3: X-Ray View of Main Processor Unit

Referring to figure 5-3 — Underneath the main processor unit cover and cooling tunnel sits the main processor package. Diving deeper with our x-ray machine we can see beneath the packaging and make out the Apple M1 Max processor *System-on-a-Chip* or *SoC* as shown in figure 5-4.

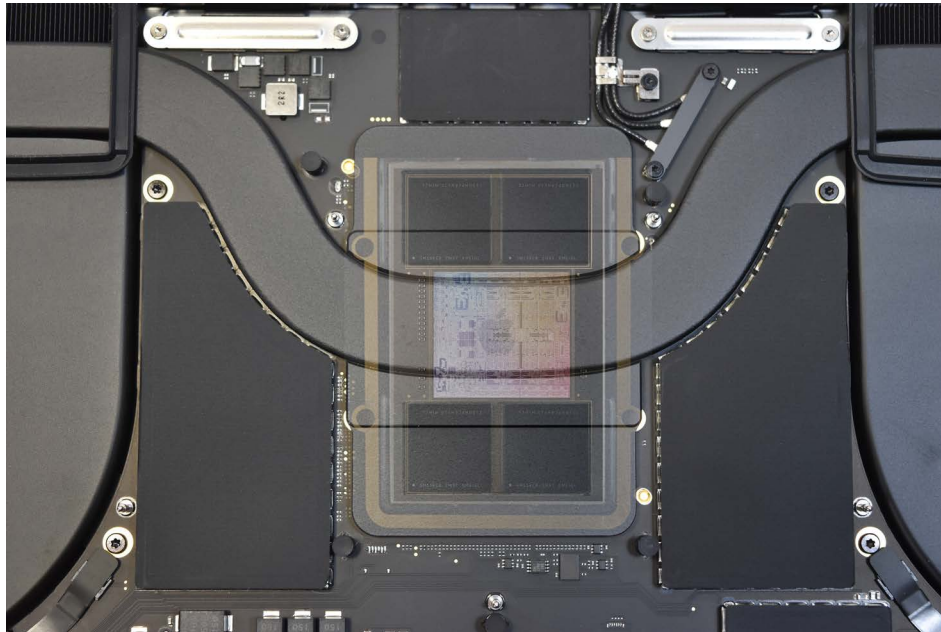


Figure 5-4: Apple M1 Max Processor System-on-a-Chip (SoC) Die

Referring to figure 5-4 — The large pale square in the center of the SoC is the main processing unit. The four darker squares are 64 gigabytes of main memory servicing the processor via Apple’s *Unified Memory Architecture* or *UMA*. Figure 5-5 zooms in closer to the M1 Max SoC.

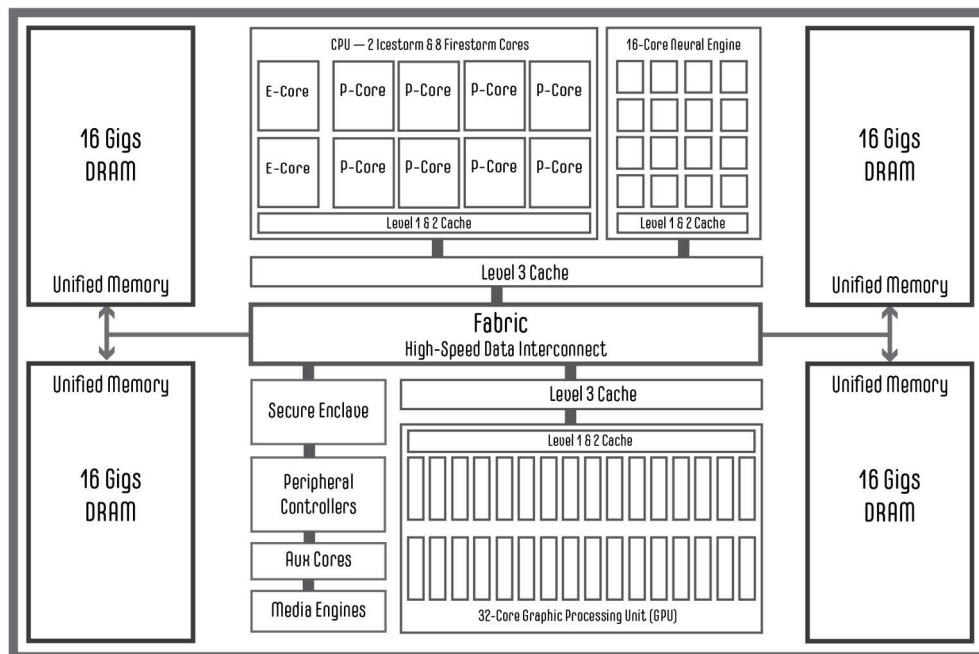


Figure 5-5: Apple M1 Max<sup>®</sup> SoC Block Diagram (*Notional*)



Referring to figure 5-5 — The M1 Max SoC contains a 10-core Central Processing Unit (CPU) with 8 performance cores (Firestorm<sup>®</sup> P-cores) and 2 efficiency cores (Icestorm<sup>®</sup> E-cores), a 32-core Graphics Processing Unit (GPU), and a 16-core Neural Engine (Neural Processing Unit or NPU). (*Firestorm and Icestorm sound like characters from an X-Men movie.*) In addition, it incorporates hardware accelerated video processing, encode, and decode engines. (See *Apple's M1 Max technical specifications* for more details.)

Main (*unified*) memory is shared between processing units, which significantly speeds memory access for image, video, and machine learning tasks. In addition to main memory, the M1 Max chip contains several levels of on-board cache memory and additional processor cores to off-load peripheral device communications and other routine processor tasks. This frees up the main CPU cores for serious computational tasks. Needless to say, this is a powerful processor for a laptop by any measure.

## 1.2 THREE ASPECTS OF PROCESSOR ARCHITECTURE

There are three aspects of processor architecture programmers should be aware of: *feature set*, *feature set implementation*, and *feature set accessibility*.

### 1.2.1 FEATURE SET

A processor's feature set derives from its design. Can floating point arithmetic be executed in hardware or must it be emulated in software? Must all data pass through the processor, or can input/output be handled off-chip while the processor goes about its business? How much memory can the processor access? How fast can it run? How much data can it process per unit time? A processor's design addresses these and other feature-set issues.

### 1.2.2 FEATURE SET IMPLEMENTATION

Feature set implementation primarily determines how a processor's functionality is arranged and executed in hardware. How does the processor implement the feature set? Is it a Reduced Instruction Set Computer (RISC) or a Complex Instruction Set Computer (CISC)? Is it superscalar and pipelined? Does it use Out-of-Order instruction execution? Does it have a vector execution unit? Is the floating-point unit on the chip with the processor, or does it sit off to the side? Is the super fast cache memory part of the processor, or is it located on another chip? Is it a multi-core processor? If so, does it support hyperthreading? If not, how does it support multiple threads of execution? These questions all deal with how processor functionality is achieved and how its design is executed.

### 1.2.3 FEATURE SET ACCESSIBILITY

Feature set accessibility is the aspect of a processor's architecture you are most concerned with as a programmer. Processor designers make a processor's feature set available to programmers via the processor's instruction set. A valid instruction in a processor's raw instruction set is a set of voltage levels that, when decoded by the processor, have special meaning. A high voltage is usually translated as "on" or "1", and a low voltage is usually translated as "off" or "0". A set of on-and-off voltages is conveniently represented to humans as a string of ones and zeros. Instructions in this format are generally referred to as *machine instructions* or *machine code*. As proces-

processor power increases, the size of machine instructions grows as well, making it extremely difficult for programmers to deal directly with machine code.

Modern processors, like the M1 Max, implement what's known as an *Instruction Set Architecture* or ISA. The ISA is an abstract model of the computer and serves as the programmer's interface to the processor. How a processor fulfills the contract specified by the ISA is a matter of implementation.

Below the ISA sits a layer of microcode that's used to sequence machine code instructions. Microcode is closely associated with the hardware in that it is burned into the chip itself or into EPROM (Erasable Programmable Read-Only Memory) and can be updated or patched. This is usually done to fix bugs detected in a processor after release.

## 1.2.4 FROM MACHINE CODE TO ASSEMBLY LANGUAGE

To make a processor's instruction set easier for humans to understand and work with, each machine instruction is represented symbolically in a set of instructions referred to as assembly language. To the programmer, assembly language represents an abstraction layer between programmer and machine intended to make the act of programming more efficient. Programs written in assembly language must be translated into machine instructions before execution. A program called an *assembler* translates assembly language into machine code.

Although assembly language is easier to work with than machine code, it requires a lot of effort to crank out an assembly language program. Assembly language programmers must busy themselves with issues like register usage and stack conventions. If you can program in assembly, you understand computer architecture.

High-level programming languages add yet another layer of abstraction. Python, with its object-oriented language features, lets programmers think in terms of solving the problem at hand, not in terms of the machine code it's ultimately executing. You will, however, need to be concerned with GPUs and Neural Engines, especially if you're doing graphics or machine learning programming

## 1.2.5 HOW THIS RELATES TO THE APPLE M1 MAX

The M1 Max is more than just a processor. It is, as you learned earlier, a System-on-a-Chip with a CPU, GPU, Neural Engine, and many more supporting services. The supporting services you can often ignore when programming, but to take full advantage of the processor's capabilities, you'll need to consider how your code will run on the CPU, the GPU, and the Neural Engine. The CPU offloads processing tasks to these specialized units as long as the code is written to take advantage of them. Apple provides libraries and tools to help developers target the M1 architecture. These include the *Metal Shading Language (MSL)* for the GPU and *Core ML* and *Core ML Tools* for the Neural Engine.

## QUICK REVIEW

A computer is a changeable machine. It's behavior is controlled by a set of instructions called a program. It's often difficult for novices to separate the notion of a computer system from the chip upon which the computer actually resides.

A computer system includes a system unit or housing, display, keyboard, mouse or trackpad, speakers, camera, microphone, and other peripheral devices. The system unit houses various

internal components including a power supply and/or battery, antennas for WiFi and Bluetooth, cooling fans, and a system board which contains the processor housing.

The real work of a computer system is performed by its processor. Modern computer systems have complex processors that are themselves considered a System-on-a-Chip (SoC). The Apple M1 Max is an SoC that contains a Central Processing Unit (CPU), Graphics Processing Unit (GPU) and a Neural Engine or Neural Processing Unit (NPU) supported by a Unified Memory Architecture (UMA) connected via a high-speed data transfer Fabric.

It's not enough to simply target the CPU with general program code. To gain full advantage of modern SoC processors requires optimized code. Apple provides Metal Shading Language (MSL) for the GPU and Core ML and Core ML Tools for the Neural Engine.

---

## 2 MEMORY ORGANIZATION

---

Most modern computers share similar memory systems design. As a programmer, you should be aware of how computer memory is organized and accessed. The best way to learn how your computer works is to poke around in memory and see what's in there for yourself. This section provides a brief introduction to computer memory concepts to help get you started.

### 2.1 MEMORY BASICS

A computer's memory stores information in the form of electronic voltages. There are two general types of memory: volatile and non-volatile. Volatile memory stores data as long as it has power. It will lose stored data if power is removed for any length of time. Main memory and cache memory, two forms of random access memory (RAM), are examples of volatile memory. Read-only memory (ROM) and auxiliary storage devices such as Blu-ray disks, CD-ROMs, DVDs, hard disk drives, USB flash drives, floppy disks, SD cards, and tapes are examples of non-volatile memory. Non-volatile memory stores data indefinitely, even when power is removed.

#### 2.1.1 MEMORY HIERARCHY

Computer systems contain several different types of memory. These memory types range from slow and cheap to fast and expensive. The proportion of slow and cheap memory to fast and expensive memory can be viewed in the shape of a pyramid commonly referred to as the *memory hierarchy* as shown in figure 5-6.

The job of a computer system designer with regards to memory subsystems is to make them perform as if all the memory they contained were fast and expensive. They use *cache memory* to store frequently used data and instructions close to the processor, and buffer disk reads into memory to give the appearance of faster disk access. Figure 5-7 shows a block diagram of the different types of memory used in a typical computer system.



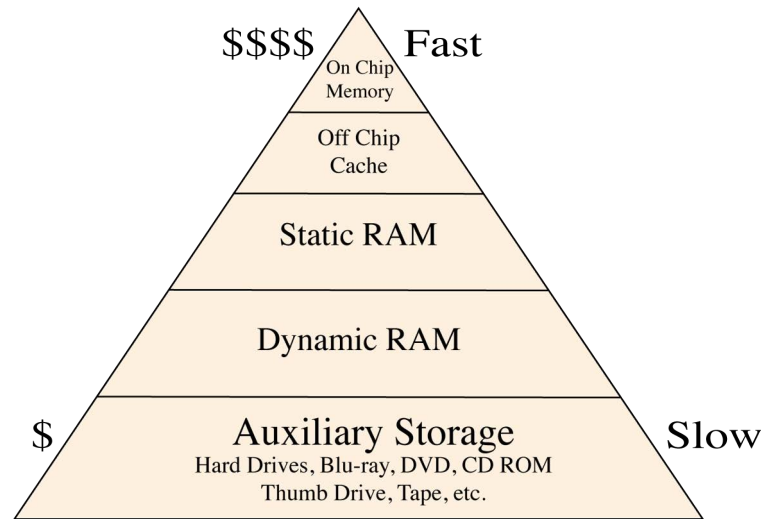


Figure 5-6: Memory Hierarchy Pyramid Showing Cost vs. Speed

During program execution, the faster cache memory is searched first by the processor for any requested data or instructions. If it's not there, a performance penalty occurs in the form of longer overall access times required to retrieve the information from a slower memory source. (A cache miss.) As chip densities grow, more cache memory is co-located on the processor chip, thus improving overall processing times. The M1 Max processor hosts multiple levels of processor and system cache memory within the SoC.

### 2.1.2 BITS, BYTES, WORDS

Program code and data are stored in main memory as electronic voltages. Since I'm talking about digital computers, the voltage levels represent two discrete states depending on the level. Usually, low voltages represent no value, off, or 0, while a high voltage represents on, or 1.

When data is stored on auxiliary memory devices, electronic voltages are translated into either electromagnetic fields (tape drives, floppy and rotating hard disks), stored voltage states (NAND flash memory solid state drives (SSDs)), or bumps that can be detected by laser beam (CDs, DVDs, etc.)

#### 2.1.2.1 BIT

The bit represents one discrete piece of information stored in a computer. On most modern computer systems bits cannot be individually accessed from memory. However, after the byte to

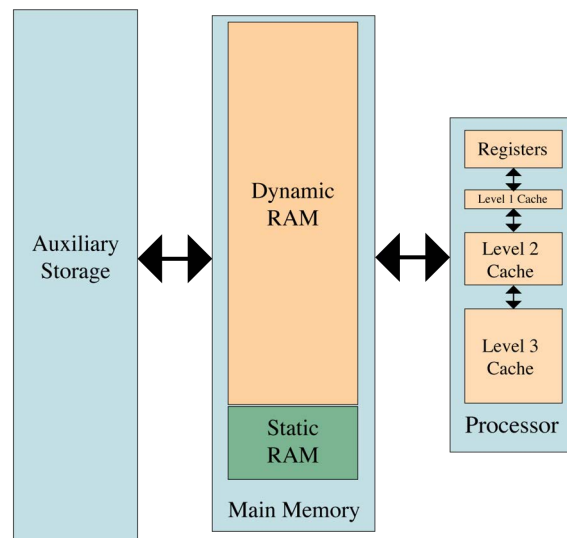


Figure 5-7: Simplified Memory Subsystem Diagram

0  
0  
0  
0  
0  
1  
0  
1

which a bit belongs is loaded into a processor register, the byte can be manipulated to access a particular bit.

### 2.1.2.2 BYTE

A byte contains 8 bits. Most computer memory is byte addressable, although as processors become increasingly powerful and can manipulate wider memory words, loading bytes by themselves into the processor becomes increasingly inefficient. For that reason, the fastest memory reads can be done a word at a time.

### 2.1.2.3 WORD

A word is a collection of bytes. The number of bytes that comprise a word is computer-system dependent. If a computer's data bus is 64 bits wide and its processor's registers are 64-bits wide, then the word size would be 8 bytes long ( $64 \text{ bits} / 8 \text{ bits} = 8 \text{ bytes}$ ). Bigger computers will have larger word sizes. This means they can manipulate more information per unit time than a computer with a smaller word size.

## QUICK REVIEW

Computer systems contain a mix of fast, expensive memory, and slow, inexpensive memory. Computer system designers must balance the use of each type of memory and structure the memory sub-system in a way that makes the computer perform as if the entire system was filled with fast, expensive memory.

Cache memory is high-speed memory located close to the processor. Modern processors contain level 1, 2, and 3 cache either on the same chip as the processor core, or within the same processor package.

A program must be fetched from auxiliary storage and loaded into main memory prior to execution. Recently accessed instructions and data are stored in cache memory for faster retrieval. A cache hit occurs when the processor finds what it's looking for in the cache. Conversely, if the required data or instruction is not found in the cache, a cache miss occurs instead, delaying program execution while the processor waits while the needed data is fetched from slower main memory.

A bit represents a voltage within the processor and is either on or off. A 1 represents on; a 0 represents off. A series of eight bits is called a byte. Multiple bytes together represent a word, and the length of a word is dictated by the type of processor and width of the memory bus. A 64-bit computer would have a word size of 64 bits or 8 bytes. Memory is read into the processor a word at a time.

---

## 3 WHAT IS A PROGRAM?

---

Intuitively you already know the answer to this question. A program is something that runs on a computer. This simple definition works well enough for most purposes, but as a programmer you will need to arm yourself with a better understanding of exactly what makes a program a program. In this section I discuss programs from two perspectives: the computer and the human. You

will find this information extremely helpful and it will tide you over until you take a formal course on computer architecture.

## 3.1 TWO VIEWS OF A PROGRAM

A program is a set of programming language instructions plus any data the instructions act upon or manipulate. This is a reasonable definition if you are a human, but if you are a processor, it will just not fly. That's because humans are great abstract thinkers and computers are not, so it is helpful to view the definition of a program from two points of view.

### 3.1.1 THE HUMAN PERSPECTIVE

Humans are the masters of abstract thought; it is the hallmark of our intelligence. High-level, object-oriented languages give us the ability to analyze a problem abstractly and symbolically express its solution in a form that is both understandable by humans and readable by other programs. By other programs, I mean the code a programmer writes must be translated from source code instructions into machine instructions recognizable by a particular processor. This translation is performed by running a compiler that converts the high-level language code into object code that targets a particular processor. The object code must then be transformed in such a way that it can be loaded from auxiliary memory into main memory for execution. Programming languages that are compiled into object code targeting specific processors are said to be *close to the metal*. These include languages like Fortran, C, C++, Haskell, Rust, and Go just to name a few.

Programming languages like Java, C#.Net, and Python are compiled into an intermediate language or bytecode that is then executed in a virtual machine runtime environment. For Java and C# the compilation step is separate and distinct from the execution step. The Python interpreter reads source files and compiles them into bytecode as part of the execution process. I'll talk more about this later.

To a programmer using a high-level programming language like Python, a program is a collection of classes that model the behavior of real-world objects in a particular problem domain. These classes model object behavior by defining object attributes (data) and methods to manipulate these attributes. On an even higher level, a program can be viewed as an interaction between objects. Functional programming adds its own twist on taming conceptual complexity. This view of a program is convenient for humans.

### 3.1.2 THE COMPUTER PERSPECTIVE

From a computer's perspective, a program is simply machine instructions and data. Usually both the instructions and data reside in the same memory space. This is referred to as a *Von Neumann architecture*. In order for a program to run, it must first be loaded into main memory. The processor must then fetch the address of the first instruction, at which point execution begins. In the early days of computing, programs were coded into computers by hand and then executed. Nowadays, all the nasty details of loading programs from auxiliary memory into main memory are handled by an *operating system* — which, by the way, is a program.

Since both instructions and data reside in main memory, how does a computer know when it is dealing with an instruction or with data? The answer to this question will be discussed in detail shortly, but here's a quick answer: it depends on what the computer is expecting. If a computer reads a memory location expecting to find an instruction and it does, everything runs fine. The

processor decodes and executes the instruction. If it reads a memory location expecting to find an instruction but instead finds garbage, then the decode fails and causes an error! Ever seen the blue screen of death in older Microsoft Windows operating systems? Modern operating systems avoid such catastrophic system failures by running application processes in dedicated memory spaces. An application crash may kill the application but will not affect other applications running on the processor.

## QUICK REVIEW

A program is a set of programming language instructions plus any data the instructions act upon or manipulate.

To a programmer using a programming language like Python, a program is a collection of classes that model the behavior of objects in a particular problem domain. These classes model object behavior by defining object attributes (data) and methods to manipulate these object attributes. On an even higher level, a program can be viewed as an interaction between objects.

From a computer's perspective, a program is simply machine instructions and data. Usually both the instructions and data reside in the same memory space.

---

## 4 THE PROCESSING CYCLE

---

Computers are powerful because they can do repetitive things really fast. When a computer executes a program, it constantly repeats a series of processing steps commonly referred to as the *processing cycle*. The processing cycle consists of four primary steps: Instruction *Fetch*, Instruction *Decode*, Instruction *Execution*, and Result *Store*. The step names can be shortened to simply Fetch, Decode, Execute, and Store. Different types of processors implement the processing cycle differently, but generally all processors carry out these four processing steps in some form or another. The processing cycle is depicted in figure 5-8.

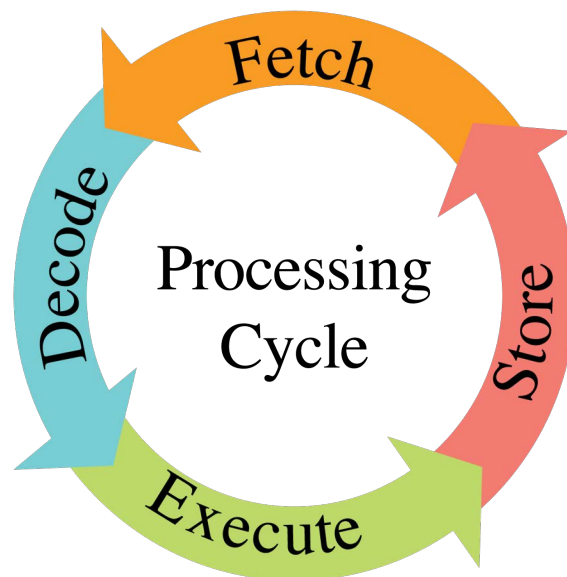


Figure 5-8: Processing Cycle

### 4.1 FETCH

In the Fetch step, the processor reads an instruction from memory and presents it to the decode section. If cache memory is present, it is checked first. Most modern processors have multiple levels of cache and separate caches for instructions and data. If the requested memory address contents resides in the cache, the read operation executes quickly (a cache hit), otherwise, the processor must wait while the data is loaded from the next level cache or from slower main memory (a cache miss). A well-designed memory subsystem minimizes processor wait times and employs various memory access prediction algorithms to maximize cache hits and minimize cache misses.

## 4.2 DECODE

In the Decode step, the fetched instruction is translated into voltages that set-up the computer's circuits for the particular operation at hand. If the computer thinks it is getting an instruction but instead gets garbage, there will be problems. A computer system's ability to recover from such situations is generally the function of a robust operating system.

## 4.3 EXECUTE

If the fetched instruction is successfully decoded as a valid instruction in the processor's instruction set, it executes. A computer is a bunch of electronic switches. Executing an instruction means the computer's electronic switches are turned either on or off to carry out the actions required by a particular instruction. Different instructions cause different sets of switches to be turned on or off. Instruction execution speed is a function of processor design and system clock speed.

## 4.4 STORE

When an instruction executes, the results, if any, must be stored somewhere. Most arithmetic instructions leave the result in one of the processor's onboard registers. Memory-write instructions would then be used to transfer these results to main memory. Keep in mind that there is only so much storage space inside a processor. At any given time, almost all data and instructions reside in main or cache memory and are only loaded into the processor when needed.

## 4.5 WHY A PROGRAM CRASHES

Notwithstanding catastrophic hardware failure, a computer crashes or locks up because what it expects to be an instruction is not. The faulty instruction loaded from memory turns out to be an unrecognizable string of ones and zeros. When it fails to decode into a proper instruction, the computer halts because of improper switch alignment.

## QUICK REVIEW

Computers are powerful because they can do repetitive things really fast. When a computer executes a program, it constantly repeats a series of processing steps commonly referred to as the processing cycle. The processing cycle consists of four primary steps: Instruction *Fetch*, Instruction *Decode*, Instruction *Execution*, and Result *Store*. The step names can be shortened to simply Fetch, Decode, Execute, and Store.

---

## 5 ALGORITHMS

---

Computers run programs; programs implement algorithms. A good working definition of an algorithm for the purpose of this book is a *recipe for getting something done on a computer*. Pretty much every line of source code you write is considered part of an algorithm. What I'd like to do in this brief section is to make you aware of the concept of good vs. better algorithms.

## 5.1 GOOD VS. BETTER ALGORITHMS

There are good ways to do something in source code and there are better ways to do the same exact thing. A good example of this can be found in the act of sorting. Suppose you want to sort in ascending order the following list of integers:

1, 10, 7, 3, 9, 2, 4, 6, 5, 8, 0, 11

One algorithm for sorting these integers might go something like this:

*Step 1:* Select the first integer position in the list

*Step 2:* Compare the selected integer with its immediate neighbor

*Step 2.1:* If the selected integer is greater than its neighbor, swap the two integers

*Step 2.2:* Else, leave it where it is

*Step 3:* Continue comparing the selected integer position with all other integers repeating steps 2.1 - 2.2

*Step 4:* Select the second integer position on the list and repeat the procedure beginning at step 2

**Continue** in this fashion **until** all integers have been compared to all other integers in the list and have been placed in their proper position.

This algorithm is simple and straightforward. It also runs pretty fast for small lists of integers, but it slows down as the list of items to be sorted grows longer. Another sorting algorithm to sort a list of integers goes as follows:

*Step 1:* Split the list into two equal sublists

*Step 2:* Repeat step 1 if any sublist contains more than two integers

*Step 3:* Sort each sublist of two integers

*Step 4:* Combine sorted sublists until all sorted sublists have been combined

This algorithm runs a little slow on small lists because of all the list splitting going on, but it sorts large lists of integers way faster than the first algorithm.

### 5.1.1 DUMB SORT

The first algorithm lists the steps for what I call a dumb sort, which is related to a *bubble sort*, but makes extra naive element comparisons. Example 5.1 gives the source code for a short program that implements the dumb sort algorithm.

5.1 dumbsort.py

```

1  """Dumb Sort Algorithm Implementation."""
2
3  import time
4  import random
5
6  class DumbSort():
7
8      def run(self, int_list):
9          innerloop = 0
10         outerloop = 0
11         swaps = 0
12
13         t_start = time.perf_counter()
14         for i in range(len(int_list)):
15             outerloop += 1
16             for j in range(1, len(int_list)):
17                 innerloop += 1

```



```

18         if int_list[j-1] > int_list[j]:
19             temp = int_list[j-1]
20             int_list[j-1] = int_list[j]
21             int_list[j] = temp
22             swaps += 1
23     t_stop = time.perf_counter()
24     sort_time = t_stop - t_start
25
26     return (int_list, outerloop, innerloop, swaps, sort_time )
27
28
29 def main():
30     ds = DumbSort()
31     list_1 = [1,10,7,3,9,2,4,6,5,8,0,11]
32     print(f'Dumb sorting {len(list_1)} integers. Don\'t blink!')
33     (sorted_list, outerloop, innerloop, swaps, sort_time) = ds.run(list_1)
34     print(f'Outerloop: {outerloop}')
35     print(f'Innerloop: {innerloop}')
36     print(f'swaps: {swaps}')
37     print(f'time: {sort_time:0.8f}')
38     print()
39     list_2 = [0,1,2,3,4,5,6,7,8,9,10,11]
40     (sorted_list, outerloop, innerloop, swaps, sort_time) = ds.run(list_2)
41     print(f'Outerloop: {outerloop}')
42     print(f'Innerloop: {innerloop}')
43     print(f'swaps: {swaps}')
44     print(f'time: {sort_time:0.8f}')
45     print()
46     list_3 = [11,10,9,8,7,6,5,4,3,2,1,0]
47     (sorted_list, outerloop, innerloop, swaps, sort_time) = ds.run(list_3)
48     print(f'Outerloop: {outerloop}')
49     print(f'Innerloop: {innerloop}')
50     print(f'swaps: {swaps}')
51     print(f'time: {sort_time:0.8f}')
52     print()
53
54     unsorted_ints = [random.randint(0, 10000) for _ in range(20000)]
55     print(f'Dumb sorting {len(unsorted_ints):,} \
56 randomly-generated integers. This may take a while. Know any good jokes?')
57     (sorted_list, outerloop, innerloop, swaps, sort_time) = ds.run(unsorted_ints)
58     print(f'Outerloop: {outerloop:,}')
59     print(f'Innerloop: {innerloop:,}')
60     print(f'swaps: {swaps:,}')
61     print(f'time: {sort_time:0.8f}')
62
63
64 if __name__ == '__main__':
65     main()
66

```

Referring to example 5.1 — The main dumb sort algorithm is implemented on lines 14 through 22. The rest is fluff, preparation, and metrics. When sorting completes, the `run()` method returns a *tuple* consisting of five items: `sorted_list`, `outerloop`, `innerloop`, `swaps`, and `sort_time`. I calculate the sort time with the help of the `time.perf_counter()` function. I calculate the sort time in the body of the `run()` method because I want to measure elapsed time as close to the sort algorithm implementation as possible so that I don't measure things like the time it takes to call the `run()` method itself, or the `print()` methods, etc.

Referring to the `main()` method — I first create a `DumbSort` object and assign it to the variable named `ds`. I create a list of 12 integers in no particular order, sort it, and report the results. I repeat this two more times, once with the list completely ordered in ascending order and again with the list completely unsorted. Finally, on lines 54 through 61, I generate a list of 20,000 random integers, sort it, and report the results.

To run the `bubblesort.py` program, clone the book's code repository, launch a terminal, navigate to the `chapter05/dumbsort` directory and use the appropriate python command for your system. I'm running macOS so I use the `python3` command like so:

```
python3 -O bubblesort.py
```

The `-O` (capital letter O) switch sets the global `__debug__` constant to `False`. To see more program output metrics launch the program with:

```
python3 bubblesort.py
```

Figure 5-9 shows the results of running this program with `__debug__` set to `False`.

```

Sat Jan 14 09:54:38 EST 2023
~/dev/cst_with_python_1st_ed/chapter05/dumbsort (main)
[513:13] swodog@macos-mojave-test-bed $ python3 -O dumbsort.py
Dumb sorting 12 integers. Don't blink!
Outerloop: 12
Innerloop: 132
swaps: 30
time: 0.00004459

Outerloop: 12
Innerloop: 132
swaps: 0
time: 0.00003076

Outerloop: 12
Innerloop: 132
swaps: 66
time: 0.00005098

Dumb sorting 20,000 randomly-generated integers. This may take a while. Know any good jokes?
Outerloop: 20,000
Innerloop: 399,980,000
swaps: 100,336,634
time: 143.68727006

```

Figure 5-9: Results of Running the Dumb Sort Program

Referring to figure 5-9 — For a list of twelve integers the outer loop executes 12 times and the inner loop executes 132 times. The first list required 30 swaps to put things in sorted order. The second list, which was already sorted, required zero swaps, while the third list required 66 swaps. Note that each list contained twelve integers which is why the inner loops and outer loops execute the same number of times for each of the three 12-item lists. The number 132 comes from the number of times the outer loop needs to execute (12) times the number of times the inner loop needs to execute ( $12 - 1 = 11$ ) for  $(12 \times 11) = 132$ . This equates to roughly  $N \times N$  where  $N$  is the number of elements in the list. For the list of 20,000 integers, the inner loop must execute  $20,000 \times 19,999 = 399,980,000$  or  $\sim 400K$  times. Each comparison and swap takes constant time which does consume time but as the number of elements that need to be sorted grows, the dominating factor in determining how long the dumb sort code will take to sort a list is determined by the

number of elements in the list squared or  $n^2$ . In computer science this is referred to as the asymptotic runtime of the algorithm and is expressed in *order notation* as  $O(n^2)$ . This may seem awful but at least the dumb sort program runs in *polynomial time*. It may take a while but it will eventually sort a big list of items. Let's look now at the second algorithm discussed earlier.

### 5.1.2 MERGE SORT

The second algorithm described at the beginning of this section is called *merge sort*. Example 5.2 give the code for an implementation of the merge sort algorithm.

5.2 *mergsort.py*

```

1  """Implements merge sort algorithm."""
2
3  import time
4  import random
5
6  class MergeSort():
7
8      def __init__(self):
9          self.callcount = 0
10
11
12     def split(self, intlist, left, right):
13         self.callcount += 1
14
15         if __debug__:
16             print(f'split() call: {self.callcount}')
17             print(intlist)
18             print(f'left: {left}')
19             print(f'right: {right}')
20             input('Press any key to continue...')
21
22         mid = 0
23         sorted_ints = None
24
25         if right > left:
26             mid = int((right + left) / 2)
27             if __debug__:
28                 print(f'mid: {mid}')
29             self.split(intlist, left, mid)
30             self.split(intlist, (mid + 1), right)
31             sorted_ints = self.merge(intlist, left, (mid + 1), right)
32
33         return sorted_ints
34
35
36     def merge(self, intlist, left, mid, right):
37
38         temp = [0] * len(intlist)
39         left_end = (mid - 1)
40         temp_pos = left
41         num_elements = (right - left + 1)
42
43         if __debug__:
44             print('merge()...')
45             print(temp)
46
47         while (left <= left_end) and (mid <= right):

```

0  
0  
0  
0  
0  
1  
0  
1

```

48         if intlist[left] <= intlist[mid]:
49             temp[temp_pos] = intlist[left]
50             temp_pos += 1
51             left += 1
52         else:
53             temp[temp_pos] = intlist[mid]
54             temp_pos += 1
55             mid += 1
56
57     while left <= left_end:
58         temp[temp_pos] = intlist[left]
59         temp_pos += 1
60         left += 1
61
62     while mid <= right:
63         temp[temp_pos] = intlist[mid]
64         temp_pos += 1
65         mid += 1
66
67     if __debug__:
68         print(f'num_elements = {num_elements}')
69
70     for i in range(num_elements):
71         intlist[right] = temp[right]
72         right -= 1
73
74     return intlist
75
76
77 def main():
78     ms = MergeSort()
79     short_list = [11,10,9,8,7,6,5,4,3,2,1,0]
80     print(f'Merge sorting {len(short_list)} unsorted integers.')
81     if __debug__:
82         print(short_list)
83         input('Press any key to continue...')
84
85     t_start = time.perf_counter()
86     sorted_list = ms.split(short_list, 0, len(short_list)-1)
87     t_stop = time.perf_counter()
88     sort_time = t_stop - t_start
89     if __debug__:
90         print(sorted_list)
91     print(f'Sort time for {len(short_list)} integers: {sort_time:0.8f} seconds.')
92     print()
93
94     unsorted_ints = [random.randint(0, 10000) for _ in range(20000)]
95     if __debug__:
96         print(unsorted_ints)
97         input('Press any key to continue...')
98
99     print(f'Merge sorting {len(unsorted_ints):,} random integers. \
100 Hold my beer...this won\'t take long.')
101     t_start = time.perf_counter()
102     sorted_ints = ms.split(unsorted_ints, 0, len(unsorted_ints)-1)
103     t_stop = time.perf_counter()
104     sort_time = t_stop - t_start
105     if __debug__:
106         print(sorted_ints)

```

```

107     print(f'Sort time for {len(unsorted_ints):,} integers: \
108 {sort_time:0.8f} seconds.')
109
110
111 if __name__ == '__main__':
112     main()
113

```

Referring to example 5.2 — If you're relatively new to programming it can take a while to wrap your head around what's happening in the code. First, the `split()` method calls itself recursively until the list no longer contains enough elements to split, at which point the recursive calls start to unwind and the `merge()` method is called to merge each split list. Since the `split()` method is a *recursive call*, the first call to `split()` on line 29 proceeds down the left half of the list. Once that series of recursive calls completes, the call to `split()` on line 30 executes and recursively splits the right half of the list. It makes more sense if you can see an animation and I recommend the one here at HackerEarth.com: <https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/visualize/>

Referring to the `main()` method — I start by creating an instance of `MergeSort` and then create a list of 12 unsorted integers to see how long it takes to sort it. Next, on line 94, I create a list of 20,000 random integers and sort it. Figure 5-10 shows the results of running this program.

```

Fri Jan 13 16:01:12 EST 2023
~/dev/cst_with_python_1st_ed/chapter05/mergesort (main)
[513:13] swodog@macos-mojave-test-bed $ python3 -0 mergesort.py
Merge sorting 12 unsorted integers.
Sort time for 12 integers: 0.00006226 seconds.

Merge sorting 20,000 random integers. Hold my beer...this won't take long.
Sort time for 20,000 integers: 1.38614055 seconds.

```

Figure 5-10: Results of Running Merge Sort Program

Referring to figure 5-10 — Bear in mind that since I'm now timing everything (method calls, setup, etc.) the reported sort times may skew slightly higher. The key takeaway in the comparison between dumb sort and merge sort is in how long it takes each to sort the list of 20,000 integers and you can see that merge sort is 100 times faster. Merge sort's asymptotic runtime is  $O(n \log n)$ . For a detailed runtime analysis I refer you to this excellent video: <https://www.youtube.com/watch?v=0nIPxaC2ITw>

## 5.2 ALGORITHM RUNTIME GROWTH RATE

When an algorithm's running time is a function of the size of its input, the term used to describe the time it takes to perform its job vs. the size of its input is called the *growth rate*. Figure 5-11 shows a plot of algorithms with the following growth rates:  $\log n$ ,  $n$ ,  $n \log n$ ,  $n^2$ ,  $n^3$ , and  $n^n$ .

Referring to figure 5-11 — You can see from the chart that dumb sort runs pretty slow and merge sort performs better. An algorithm with a growth rate of  $n$  is said to run in linear time, while an algorithm with a growth rate of  $n^n$  is said to have an exponential runtime.

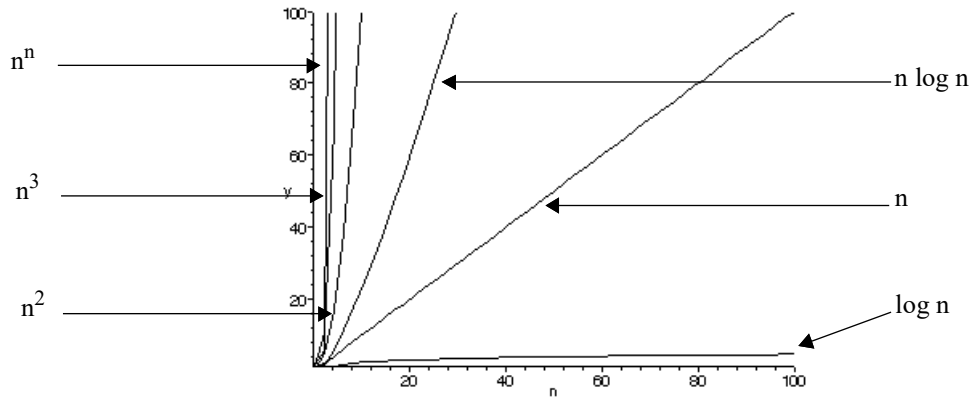


Figure 5-11: Algorithmic Growth Rates

### QUICK REVIEW

Computers run programs; programs implement algorithms. A good working definition of an algorithm for the purpose of this book is a recipe for getting something done on a computer. Pretty much every line of source code you write is considered part of an algorithm.

## 6 HOW PYTHON RUNS PROGRAMS

In this section I want to show you how Python executes programs and modules.

### 6.1 PYTHON EXECUTION PROCESS

The overall process used by the Python interpreter to execute programs is depicted in figure 5-12.

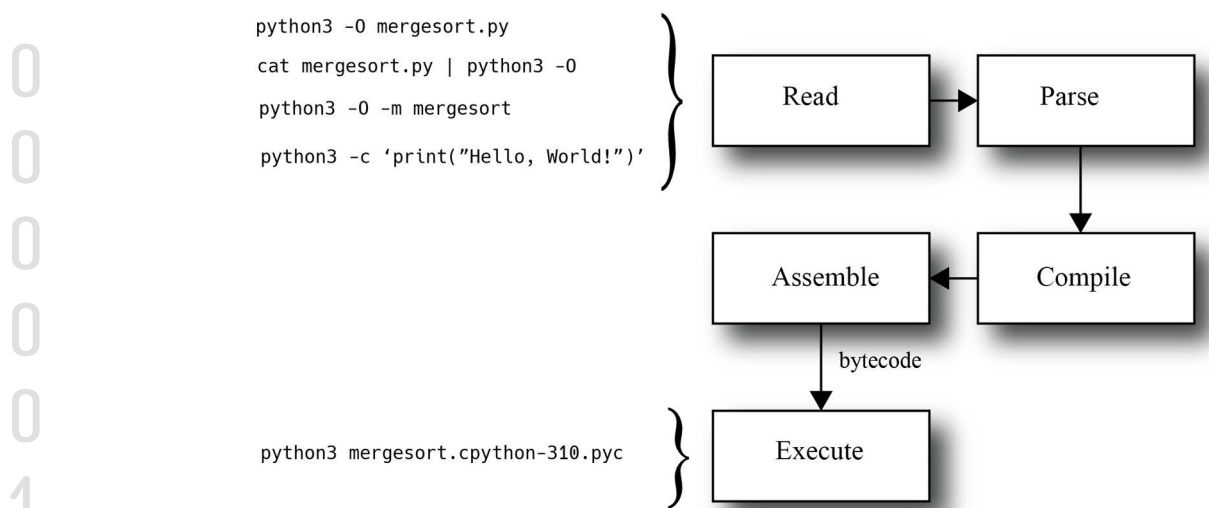


Figure 5-12: Python Interpreter Process

0  
0  
0  
0  
0  
0  
1  
0  
1



Referring to figure 5-12 — The upper left section of the diagram shows several ways Python source files and strings can be loaded into the interpreter for execution. I'll discuss each of these in more detail a bit later.

When a plain text Python source file or string is run from the command line or loaded into the interactive interpreter it is read in and presented to the parser. Any syntax errors present in the code will cause the parser to exit with a *SyntaxError*. (See Chapter 3 page 99) Actually, you don't need to see Chapter 3 page 99 because if you're learning to program, you'll see all sorts of *SyntaxError* messages.

OK, if your code makes it past the parser it goes to the compiler and on to the assembler where it comes out as *bytecode* at which point it may be saved to disk as a code object (.pyc) and/or passed to the evaluation loop for execution.

The lower left section of the diagram shows a code object (.pyc) file being run directly from the command line. Let's step through each of these methods of running a python file and note the results.

## 6.2 COMMAND-LINE SCRIPT EXECUTION

You've been exposed to this way of executing Python programs from the command line throughout the book. If a Python script is configured to run as a main module with the now familiar construct...

```
if __name__ == '__main__':
    main()
```

...you can run it from the command line like so:

```
python3 -0 mergesort.py
```

In this case I'm executing the mergesort.py script with the `-0` switch, which, as you learned earlier, sets the global constant `__debug__` to `False`. Figure 5-13 shows the results.

```

-bash
~/d/c/c/mergesort | main - * 1/18, 8:31 AM
Wed Jan 18 08:29:10 EST 2023
~/dev/cst_with_python_1st_ed/chapter05/mergesort (main)
[527:27] swodog@macos-mojave-test-bed $ python3 -0 mergesort.py
Merge sorting 12 unsorted integers.
Sort time for 12 integers: 0.00006558 seconds.

Merge sorting 20,000 random integers. Hold my beer...this won't take long.
Sort time for 20,000 integers: 1.46706126 seconds.

Wed Jan 18 08:29:32 EST 2023
~/dev/cst_with_python_1st_ed/chapter05/mergesort (main)
[528:28] swodog@macos-mojave-test-bed $ dir
total 8
drwxr-xr-x  3 swodog  staff   96 Jan 18 08:28 .
drwxr-xr-x  4 swodog  staff  128 Jan 14 09:46 ..
-rw-r--r--  1 swodog  staff 2482 Jan 14 10:04 mergesort.py

```

Figure 5-13: Command Line Script Execution

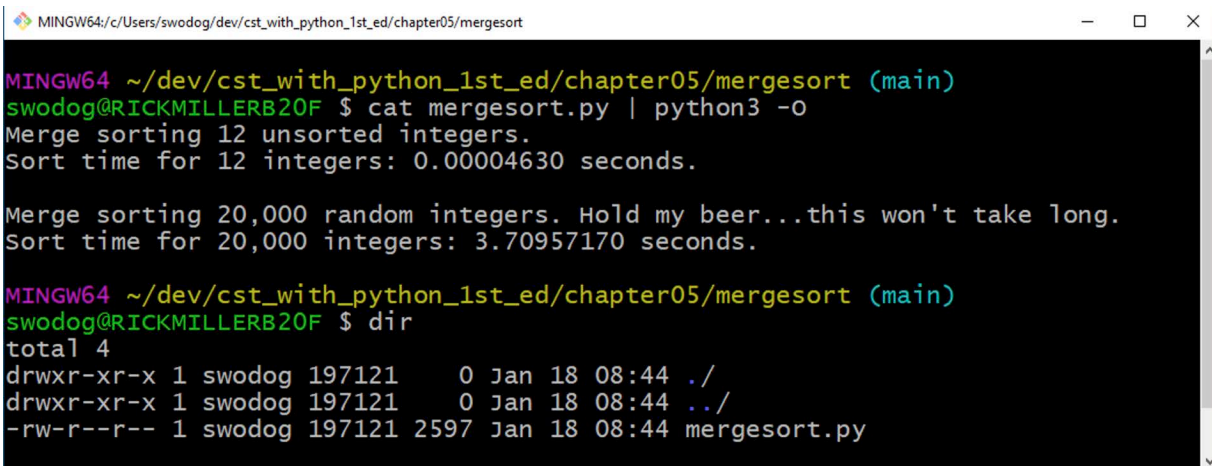
Referring to figure 5-13 — You've seen this earlier in the chapter. When the script completes, I list the contents of the mergesort directory to show no code object files have been created as a result of running the script.

## 6.3 PIPE SCRIPT TO PYTHON COMMAND

Another way to run a Python script from the command line is to pipe the output from a command to the Python interpreter as this code snippet shows:

```
cat mergesort.py | python3 -0
```

The `cat` command normally writes the contents of a file to `stdout` but the pipe command `|` redirects the output to the `stdin` of the `python3` command. Figure 5-14 shows the effects of running this command in a Git Bash terminal on Windows.



```
MINGW64 ~/dev/cst_with_python_1st_ed/chapter05/mergesort (main)
swodog@RICKMILLERB20F $ cat mergesort.py | python3 -0
Merge sorting 12 unsorted integers.
Sort time for 12 integers: 0.00004630 seconds.

Merge sorting 20,000 random integers. Hold my beer...this won't take long.
Sort time for 20,000 integers: 3.70957170 seconds.

MINGW64 ~/dev/cst_with_python_1st_ed/chapter05/mergesort (main)
swodog@RICKMILLERB20F $ dir
total 4
drwxr-xr-x 1 swodog 197121  0 Jan 18 08:44 ./
drwxr-xr-x 1 swodog 197121  0 Jan 18 08:44 ../
-rw-r--r-- 1 swodog 197121 2597 Jan 18 08:44 mergesort.py
```

Figure 5-14: Pipe Python Script To Python Interpreter

Referring to figure 5-14 — Again, this method of running a Python script produces no code object files.

## 6.4 COMMAND-LINE MODULE EXECUTION

Some clarification is in order here before proceeding. The `mergesort.py` file is considered a script because it can be directly executed by the Python interpreter at the command line. Remember, when a file is run on the command line via the `python` or `python3` command, its `__name__` property is implicitly set to `'__main__'` indicating it is the main module. If instead a Python source file is intended to be imported by another Python file then it is referred to as a module and its `__name__` property is set to its filename sans the `.py` extension. For example, if `mergesort` were imported vs. directly executed then its `__name__` property would be set to `'mergesort'`. All Python files loaded into the interpreter are transformed into modules.

### 6.4.1 EXPLICITLY SETTING MODULE `__NAME__` TO `'__MAIN__'`

The following command will execute a Python module and explicitly set its `__name__` property to `'__main__'`:

```
python3 -0 -m mergesort
```

Figure 5-15 shows the results of running this command.

Referring to figure 5-15 — The primary difference between running `mergesort` as a module vs. a script is that code object files are generated in the `__pycache__` directory. Also, the Python `sys.path` is searched. Let's explore the `__pycache__` directory. Figure 5-16 gives the listing.

```

Wed Jan 18 10:14:31 EST 2023
~/dev/cst_with_python_1st_ed/chapter05/mergesort (main)
[555:55] swodog@macos-mojave-test-bed $ python3 -O -m mergesort
Merge sorting 12 unsorted integers.
Sort time for 12 integers: 0.00006159 seconds.

Merge sorting 20,000 random integers. Hold my beer...this won't take long.
Sort time for 20,000 integers: 1.42179981 seconds.

Wed Jan 18 10:14:44 EST 2023
~/dev/cst_with_python_1st_ed/chapter05/mergesort (main)
[556:56] swodog@macos-mojave-test-bed $ dir
total 8
drwxr-xr-x  4 swodog  staff   128 Jan 18 10:14 .
drwxr-xr-x  5 swodog  staff   160 Jan 18 09:27 ..
drwxr-xr-x  3 swodog  staff    96 Jan 18 10:14 __pycache__
-rw-r--r--  1 swodog  staff  2482 Jan 14 10:04 mergesort.py

```

Figure 5-15: Command Line Module Execution — Code File Generated

```

Wed Jan 18 10:21:54 EST 2023
~/dev/cst_with_python_1st_ed/chapter05/mergesort (main)
[558:58] swodog@macos-mojave-test-bed $ dir __pycache__/
total 8
drwxr-xr-x  3 swodog  staff    96 Jan 18 10:14 .
drwxr-xr-x  4 swodog  staff   128 Jan 18 10:14 ..
-rw-r--r--  1 swodog  staff  2376 Jan 18 10:14 mergesort.cpython-310.opt-1.pyc

```

Figure 5-16: `__pycache__` Directory Listing

Referring to figure 5-16 — Executing a module generates Python *code object* files and stores them in the `__pycache__` directory. A code object file (`.pyc`) represents a compiled and assembled Python program that can be loaded into the Python interpreter and executed by the Python runtime. A code file does not contain bytecode in an editable form. In other words, you can't open a `.pyc` file in a text editor and expect to see intelligible bytecode instructions. Modules must be loaded into the Python interpreter before they can be disassembled. I'll get to that shortly. First, let's execute a code object file.

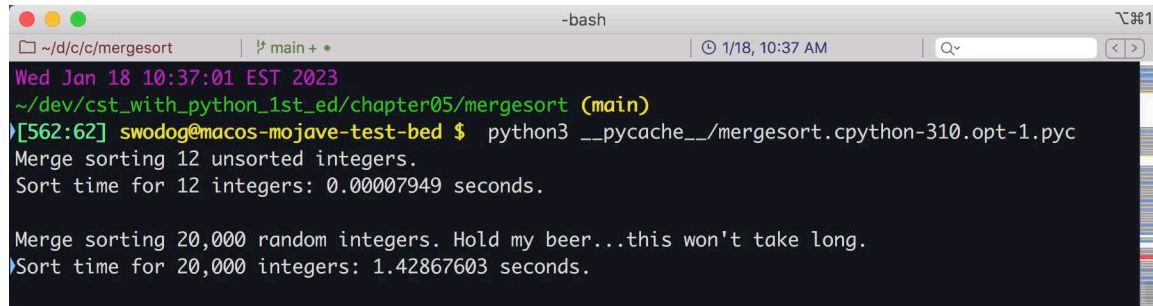
## 6.5 COMMAND-LINE CODE OBJECT FILE EXECUTION

Code object files represent Python programs that have been parsed, compiled, and assembled and can be executed without having to go again through all those steps. The purpose of generating compiled byte code files is to speed up execution of scripts and modules that haven't changed. If you do make a change to a script of module source code, it will be recompiled.

To execute a code object file just run it directly with the Python command-line tool like so:

```
python3 __pycache__/mergesort.cpython-310.opt-1.pyc
```

Note that if you're following along, the name of the code file generated when you executed the module may be different. Just F.Y.I. Figure 5-17 shows the results of running the code file.



```

-bash
~/d/c/c/mergesort | main + *
Wed Jan 18 10:37:01 EST 2023
~/dev/cst_with_python_1st_ed/chapter05/mergesort (main)
[562:62] swodog@macos-mojave-test-bed $ python3 __pycache__/mergesort.cpython-310.opt-1.pyc
Merge sorting 12 unsorted integers.
Sort time for 12 integers: 0.00007949 seconds.

Merge sorting 20,000 random integers. Hold my beer...this won't take long.
Sort time for 20,000 integers: 1.42867603 seconds.

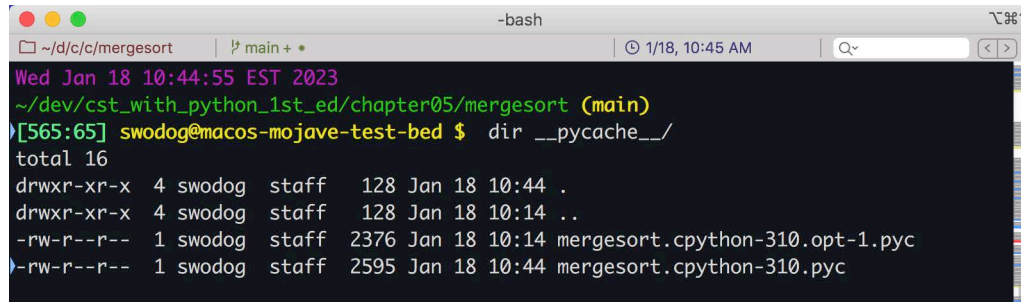
```

Figure 5-17: Command-Line Code File Execution

Referring to figure 5-17 — Note that it was unnecessary to supply the `-O` switch to set the `__debug__` constant to `False` because the code file has already been compiled. I'll execute the module again but this time omit the `-O` switch:

```
python3 -m mergesort
```

Figure 5-18 shows the `__pycache__` directory listing after running this command.



```

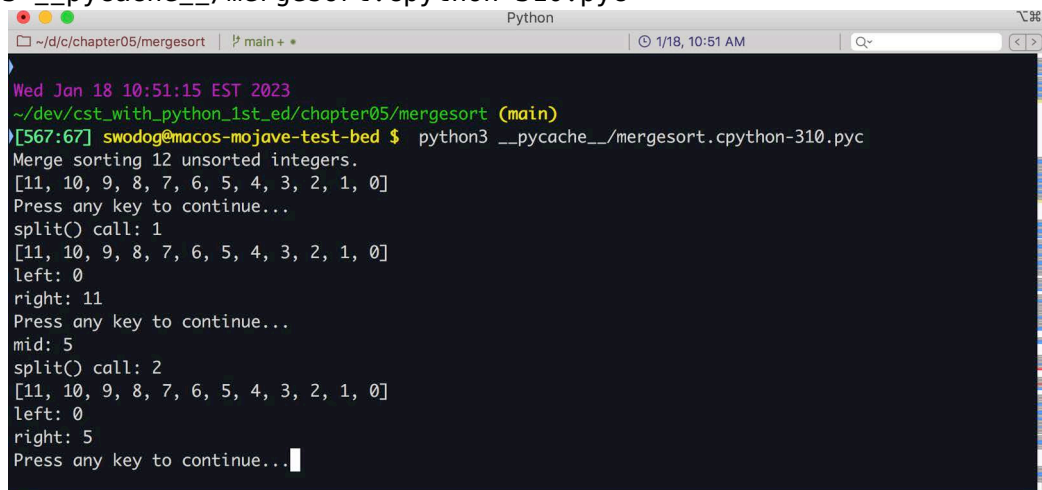
-bash
~/d/c/c/mergesort | main + *
Wed Jan 18 10:44:55 EST 2023
~/dev/cst_with_python_1st_ed/chapter05/mergesort (main)
[565:65] swodog@macos-mojave-test-bed $ dir __pycache__/
total 16
drwxr-xr-x  4 swodog  staff   128 Jan 18 10:44  .
drwxr-xr-x  4 swodog  staff   128 Jan 18 10:14  ..
-rw-r--r--  1 swodog  staff  2376 Jan 18 10:14  mergesort.cpython-310.opt-1.pyc
-rw-r--r--  1 swodog  staff  2595 Jan 18 10:44  mergesort.cpython-310.pyc

```

Figure 5-18: `__pycache__` Directory Listing — Another Code File Has Been Generated

Referring to figure 5-18 — Running the `mergesort` module without the `-O` switch has generated another version of the code file in the `__pycache__` directory. Figure 5-19 shows the partial results of running the new code file with the following command:

```
python3 __pycache__/mergesort.cpython-310.pyc
```



```

Python
~/d/c/c/chapter05/mergesort | main + *
Wed Jan 18 10:51:15 EST 2023
~/dev/cst_with_python_1st_ed/chapter05/mergesort (main)
[567:67] swodog@macos-mojave-test-bed $ python3 __pycache__/mergesort.cpython-310.pyc
Merge sorting 12 unsorted integers.
[11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
Press any key to continue...
split() call: 1
[11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
left: 0
right: 11
Press any key to continue...
mid: 5
split() call: 2
[11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
left: 0
right: 5
Press any key to continue...

```

Figure 5-19: Running Code File with `__debug__` set to `True`



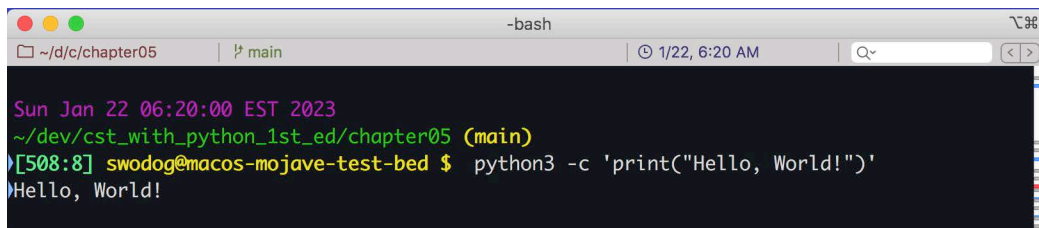
Referring to figure 5-19 — Notice that running the code file does not require the use of the `-O` switch because it has already been compiled and assembled. Again, to clarify, you don't normally need to execute a code object file directly as this is handled automatically for you by the Python interpreter.

## 6.6 COMMAND-LINE PYTHON COMMAND EXECUTION

You can also execute a Python string directly from the command line by using the `-c` switch as this code snippet shows:

```
python3 -c 'print("Hello, World!")'
```

Figure 5-20 shows the results of running this program.



```

Sun Jan 22 06:20:00 EST 2023
~/dev/cst_with_python_1st_ed/chapter05 (main)
[508:8] swodog@macos-mojave-test-bed $ python3 -c 'print("Hello, World!")'
Hello, World!

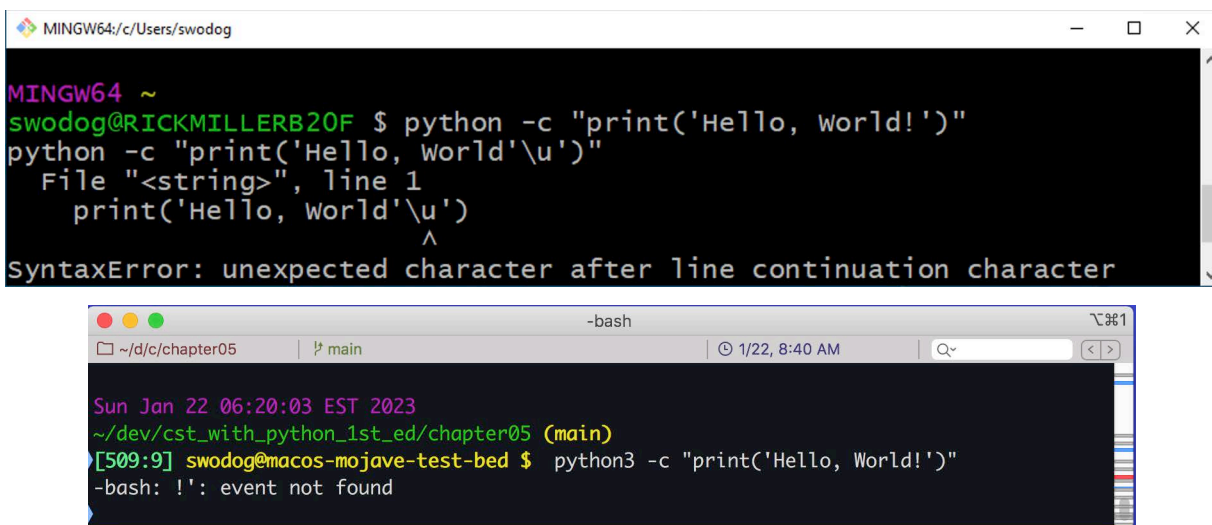
```

Figure 5-20: Running Python String with `-c` Switch

Referring to figure 5-20 — At first blush you may wonder, "What's it good for, this running Python strings with the `-c` switch?" Great question. When I find a good answer for its practical uses I'll update this book. In the meantime, just know it's a possibility but not very useful for our purposes. Another reason to avoid it here is because the placement of the quotes affects how the input string is parsed. For example, applying the quotes in this order causes an error:

```
python -c "print('Hello, World!')"
```

Figure 5-21 shows the results of executing this command.



```

MINGW64: c:/Users/swodog
MINGW64 ~
swodog@RICKMILLERB20F $ python -c "print('Hello, world!')"
python -c "print('Hello, world'\u')"
File "<string>", line 1
  print('Hello, world'\u')
                        ^
SyntaxError: unexpected character after line continuation character

```

```

Sun Jan 22 06:20:03 EST 2023
~/dev/cst_with_python_1st_ed/chapter05 (main)
[509:9] swodog@macos-mojave-test-bed $ python3 -c "print('Hello, World!')"
-bash: !': event not found

```

Figure 5-21: Errors Caused by Quote Placement Git Bash (top) and macOS (above)

Referring to figure 5-21 — The same command produces different errors depending on which operating system you're using. So, enough said about the `-c` switch for the purposes of this book.

## 6.7 DISASSEMBLING PYTHON MODULE

As you learned above, a code object file (.pyc) contains compiled Python code, however, you cannot open the file with a text editor and see intelligible bytecode instructions. To see those, you'll need to load a module into the Python interpreter and use the `dis` module to disassemble it. Let's disassemble the following program.

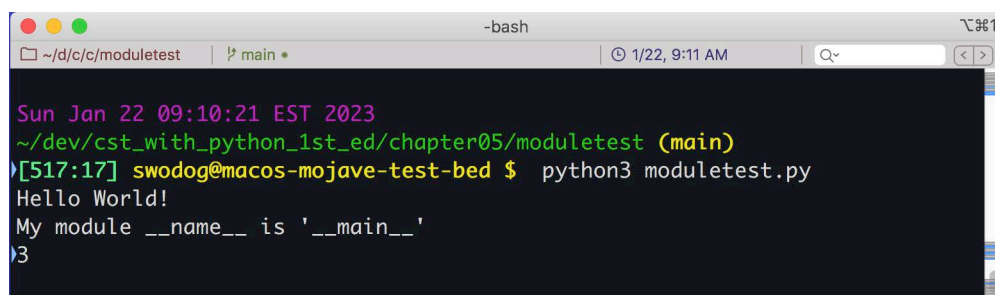
5.3 *moduletest.py*

```

1  """Test Module"""
2
3  class ModuleTest():
4
5      def __init__(self):
6          pass
7
8      def say_hi(self):
9          print('Hello World!')
10         print(f'My module __name__ is \'{__name__}\'' )
11         var_one = 1
12         var_two = 2
13         sum = var_one + var_two
14         print(f'{sum}')
15
16
17  def main():
18      mt = ModuleTest()
19      mt.say_hi()
20
21
22  if __name__ == '__main__':
23      main()

```

Referring to example 5.3 — The `moduletest` module defines a class named `ModuleTest` with two methods, `__init__()` and `say_hi()`. All the action is in the `say_hi()` method. First, it prints the string "Hello World!" to the console. On the next line it prints the value of the `__name__` property whose value will change depending on whether the module is executed vs. loaded, as you'll soon see. Finally, it creates two variables, adds them together, and prints the results. Figure 5-22 shows the results of running this program from the command line.



```

-bash
~/dev/cst_with_python_1st_ed/chapter05/moduletest | main * 1/22, 9:11 AM
Sun Jan 22 09:10:21 EST 2023
~/dev/cst_with_python_1st_ed/chapter05/moduletest (main)
[517:17] swodog@macos-mojave-test-bed $ python3 moduletest.py
Hello World!
My module __name__ is '__main__'
3

```

Figure 5-22: Running `moduletest.py` from Command Line

Referring to figure 5-22 — Note that the module's `__name__` property is set to `'__main__'` when run from the command line. Now, let's disassemble this module. To do this, launch the Python interpreter in interactive mode, import the `moduletest` and `dis` modules, and use the `dis.dis()` function to disassemble the `say_hi()` method as shown in figure 5-23.



```

Sun Jan 22 10:56:57 EST 2023
~/dev/cst_with_python_1st_ed/chapter05/moduletest (main)
[519:19] swodog@macos-mojave-test-bed $ python3
Python 3.10.8 (main, Oct 13 2022, 10:19:13) [Clang 12.0.0 (clang-1200.0.32.29)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import moduletest, dis
>>> print(moduletest.__name__)
moduletest
>>> dis.dis(moduletest.ModuleTest.say_hi)
 9          0 LOAD_GLOBAL              0 (print)
          2 LOAD_CONST                1 ('Hello World!')
          4 CALL_FUNCTION              1
          6 POP_TOP

10         8 LOAD_GLOBAL              0 (print)
         10 LOAD_CONST                2 ("My module __name__ is ")
         12 LOAD_GLOBAL              1 (__name__)
         14 FORMAT_VALUE              0
         16 LOAD_CONST                3 ('')
         18 BUILD_STRING             3
         20 CALL_FUNCTION              1
         22 POP_TOP

11         24 LOAD_CONST                4 (1)
         26 STORE_FAST              1 (var_one)

12         28 LOAD_CONST                5 (2)
         30 STORE_FAST              2 (var_two)

13         32 LOAD_FAST               1 (var_one)
         34 LOAD_FAST               2 (var_two)
         36 BINARY_ADD
         38 STORE_FAST              3 (sum)

14         40 LOAD_GLOBAL              0 (print)
         42 LOAD_FAST               3 (sum)
         44 FORMAT_VALUE              0
         46 CALL_FUNCTION              1
         48 POP_TOP
         50 LOAD_CONST                0 (None)
         52 RETURN_VALUE
>>>

```

Figure 5-23: Using `dis.dis()` Function to Disassemble `moduletest.say_hi()` Method

Referring to figure 5-23 — The disassembler output consists of three columns. The first column refers to the source code line number, so the 9 refers to line 9 of example 5.3, and on and on. The second column lists the bytecode instructions and their offset from the beginning of the function stack frame. The third column lists the operands. A complete list of bytecode instructions for Python 3.10 can be found here: <https://docs.python.org/3.10/library/dis.html#python-bytecode-instructions>.

## 6.8 USING `__MAIN__.PY` FOR APPLICATION ENTRY POINT

Up to this point in the book, I have been putting a `main()` function either in a separate, stand-alone `main.py` module or in the module itself, as is the case with the `dumbort.py`, `merge-`

`sort.py`, and `moduletest.py` scripts discussed in this chapter. An alternative method you can use to create an application entry point is to add a `__main__.py` file to a directory. Example 5.4 gives the code for a `__main__.py` file that imports and runs the `ModuleTest.say_hi()` method.

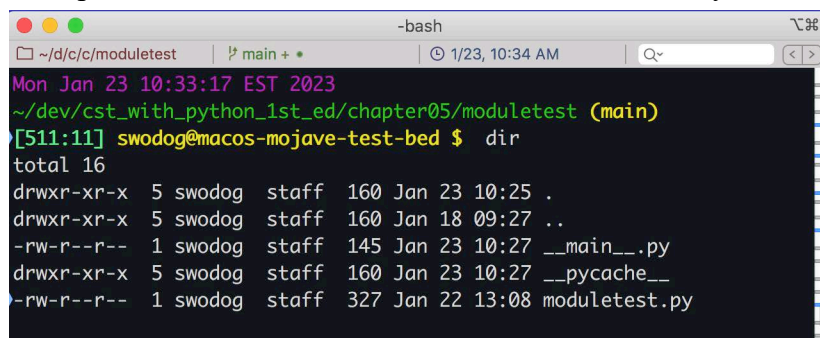
5.4 `__main__.py`

```

1  """Application Entry Point"""
2
3  from moduletest import ModuleTest
4
5  def main():
6      mt = ModuleTest()
7      mt.say_hi()
8
9  if __name__ == '__main__':
10     main()
11

```

Referring to example 5.4 — Save this file in the `moduletest` directory as shown in figure 5-24.



```

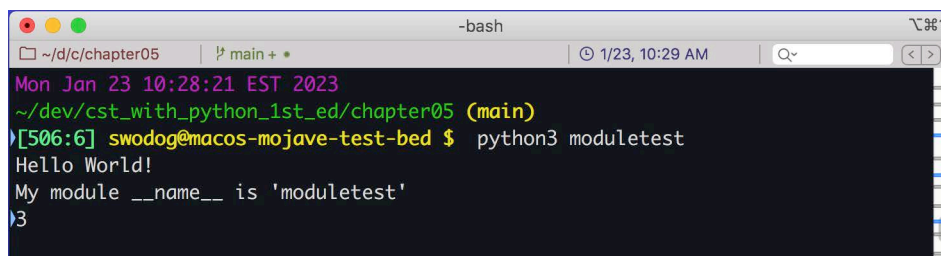
-bash
~/dev/cst_with_python_1st_ed/chapter05/moduletest (main)
[511:11] swodog@macos-mojave-test-bed $ dir
total 16
drwxr-xr-x  5 swodog  staff  160 Jan 23 10:25  .
drwxr-xr-x  5 swodog  staff  160 Jan 18 09:27  ..
-rw-r--r--  1 swodog  staff  145 Jan 23 10:27  __main__.py
drwxr-xr-x  5 swodog  staff  160 Jan 23 10:27  __pycache__
-rw-r--r--  1 swodog  staff  327 Jan 22 13:08  moduletest.py

```

Figure 5-24: `moduletest` Directory with `__main__.py` File

To run this program, move up one directory (`cd ..`) and type the following command:  
`python3 moduletest`

Figure 5-25 shows the results of running this program.



```

-bash
~/dev/cst_with_python_1st_ed/chapter05 (main)
[506:6] swodog@macos-mojave-test-bed $ python3 moduletest
Hello World!
My module __name__ is 'moduletest'
3

```

Figure 5-25: Results of Running the Code Snippet Above

## 6.9 REDIRECTING PROGRAM OUTPUT

Often times, you'll need to analyze program output but the program writes too much data to the console to do any practical analysis. To aid with offline program output analysis you can redirect the output from the `stdout` to a file as shown in the following code snippet.

```
python3 moduletest > moduletest/output.txt
```

To run this code snippet, navigate to the directory above the *moduletest* directory, run the *moduletest* program as you did in the previous section, and use the redirect character '`>`' to send the output to the *moduletest/output.txt* file. I'll leave it to you to examine the contents of the *output.txt* file.

## QUICK REVIEW

The Python program execution process consists of reading, parsing, compiling, assembling, and execution phases. There are multiple ways to run a Python program.

Scripts are source files intended to be run from the command line. Modules are source files intended to be imported by other scripts. A source file can function as a script, a module, or both. Scripts have a main entry point and their module `__name__` attribute is set to '`__main__`' if they are run directly from the command line or with the `-m` switch. Importing a module generates code object files stored in a `__pycache__` directory. The Python interpreter uses compiled code object files to speed up repeated module execution by avoiding the read, parse, compile, and assemble phases. Making a modification to a source file will cause a recompilation.

Add a `__main__.py` file to a directory for an alternative application entry point. To save program output for offline analysis, redirect output to a file with the redirect character '`>`'.

---

## SUMMARY

---

A computer is a changeable machine. It's behavior is controlled by a set of instructions called a program. It's often difficult for novices to separate the notion of a computer system from the chip upon which the computer actually resides.

A computer system includes a system unit or housing, display, keyboard, mouse or trackpad, speakers, camera, microphone, and other peripheral devices. The system unit houses various internal components including a power supply and/or battery, antennas for WiFi and Bluetooth, cooling fans, and a system board which contains the processor housing.

The real work of a computer system is performed by its processor. Modern computer systems have complex processors that are themselves considered a System-on-a-Chip (SoC). The Apple M1 Max is an SoC that contains a Central Processing Unit (CPU), Graphics Processing Unit (GPU) and a Neural Engine or Neural Processing Unit (NPU) supported by a Unified Memory Architecture (UMA) connected via a high-speed data transfer Fabric.

It's not enough to simply target the CPU with general program code. To gain full advantage of modern SoC processors requires optimized code. Apple provides Metal Shading Language (MSL) for the GPU and Core ML and Core ML Tools for the Neural Engine.

Computer systems contain a mix of fast, expensive memory, and slow, inexpensive memory. Computer system designers must balance the use of each type of memory and structure the memory sub-system in a way that makes the computer perform as if the entire system was filled with fast, expensive memory.

Cache memory is high-speed memory located close to the processor. Modern processors contain level 1, 2, and 3 cache either on the same chip as the processor core, or within the same processor package.

A program must be fetched from auxiliary storage and loaded into main memory prior to execution. Recently accessed instructions and data are stored in cache memory for faster retrieval. A

cache hit occurs when the processor finds what it's looking for in the cache. Conversely, if the required data or instruction is not found in the cache, a cache miss occurs instead, delaying program execution while the processor waits while the needed data is fetched from slower main memory.

A bit represents a voltage within the processor and is either on or off. A 1 represents on; a 0 represents off. A series of eight bits is called a byte. Multiple bytes together represent a word, and the length of a word is dictated by the type of processor and width of the memory bus. A 64-bit computer would have a word size of 64 bits or 8 bytes. Memory is read into the processor a word at a time to maximize efficiency.

A program is a set of programming language instructions plus any data the instructions act upon or manipulate.

To a programmer using a programming language like Python, a program is a collection of classes that model the behavior of objects in a particular problem domain. These classes model object behavior by defining object attributes (data) and methods to manipulate these object attributes. On an even higher level, a program can be viewed as an interaction between objects.

From a computer's perspective, a program is simply machine instructions and data. Usually both the instructions and data reside in the same memory space.

Computers are powerful because they can do repetitive things really fast. When a computer executes a program, it constantly repeats a series of processing steps commonly referred to as the processing cycle. The processing cycle consists of four primary steps: Instruction *Fetch*, Instruction *Decode*, Instruction *Execution*, and Result *Store*. The step names can be shortened to simply Fetch, Decode, Execute, and Store.

Computers run programs; programs implement algorithms. A good working definition of an algorithm for the purpose of this book is a recipe for getting something done on a computer. Pretty much every line of source code you write is considered part of an algorithm.

The Python program execution process consists of reading, parsing, compiling, assembling, and execution phases. There are multiple ways to run a Python program.

Scripts are source files intended to be run from the command line. Modules are source files intended to be imported by other scripts. A source file can function as a script, a module, or both. Scripts have a main entry point and their module `__name__` attribute is set to `'__main__'` if they are run directly from the command line or with the `-m` switch. Importing a module will generate code object files stored in a `__pycache__` directory. The Python interpreter uses compiled code object files to speed up repeated module execution by avoiding the read, parse, compile, and assemble phases. Making a modification to a source file will cause a recompilation.

Add a `__main__.py` file to a directory for an alternative application entry point. To save program output for offline analysis, redirect output to a file with the redirect character `'>'`.

---

## SKILL-BUILDING EXERCISES

---

1. **Research Your Computer:** Research your computer and its processor. Your goal is to understand its capabilities and limitations. Who makes the processor? Intel, AMD, Apple? Is it CISC or RISC-based? Is it a System-on-a-Chip (SoC)? How does it compare to other processors in its class?

2. **Compare Different Processors:** Complete a survey of the latest processors available from Intel, AMD, Apple, NVIDIA, Qualcomm, Rockchip, and Fujitsu. How does the processor you have in your laptop compare to the processors offered by these companies. Move outside your comfort zone and explore processors not found in your computer.
3. **Python Interpreter Switches:** You've seen in this chapter the use of the `-0`, `-m`, and `-c` interpreter switches. Research all the interpreter switches and note their purpose and use.
4. **Running Script vs. Module:** What are the differences between running a script directly vs. running a module with the `-m` switch?
5. **Module `__name__` Attribute:** What's the value of a module's `__name__` attribute set to when you run a script directly with the Python interpreter command-line tool.
6. **Code Object Files:** What's the purpose of Python code object files stored in the `__pycache__` directory?
7. **Syntax Errors:** During what phase of the Python execution process are syntax errors caught? What's the result of detecting a syntax error?
8. **Algorithm Growth Rates:** Research algorithm growth rates. Do there exist problems that take longer than  $n^n$  time to solve? If so, list several with a brief explanation of why the problem is so hard to solve.
9. **Using `__main__.py` As An Application Entry Point:** Research the use of `__main__.py` as an entry point. In your own words describe a situation where that would be the preferred method to launch an application.
10. **Python Interpreter Implemented In Python:** Dive deeper into how the Python interpreter works by studying this excellent article by Allison Kaptur: <http://www.aosabook.org/en/500L/a-python-interpreter-written-in-python.html>

---

## SUGGESTED PROJECTS

---

1. **Disassemble a Python Module:** Using the `dis` module discussed in this chapter, use it to disassemble a function or method you created. Keep the function small and simple so that you can easily understand the generated bytecode instructions.
2. **Dive Deeper Into Python Internals:** Obtain the excellent book titled *CPython Internals: Your Guide To The Python 3 Interpreter*, First Edition, by Anthony Shaw, ISBN: 9781775093343
3. **Buying A New Computer:** If you're in the market for a new computer, use the information you learned in this chapter to evaluate several alternatives. List their features, processors, and processor architectures. If you make a buying decision, explain your rationale for your choice.

---

## SELF-TEST QUESTIONS

---

1. Why do you think a SoC processor performs better than a typical processor?
2. What module can you use to disassemble Python code?
3. What's the purpose of the `dir()` function?
4. If you had the choice between using one of two sorting algorithms with a growth rate of  $n^3$  or  $n$ , which one would you chose and why?
5. What is meant by the term growth rate?
6. What are the phases of the Python execution process?
7. Can you run Python code object files (`.pyc`) directly?
8. What's the purpose of the Python interpreter command-line `-m` switch?
9. What makes a computer a unique device?
10. List at least five components of a typical computer system.

---

## REFERENCES

---

Python Documentation, <https://docs.python.org/3/>

A Python Interpreter Written in Python, Allison Kaptur, <http://www.aosabook.org/en/500L/a-python-interpreter-written-in-python.html>

Apple M1 vs. Intel Core I9: A Deeper Look, <https://techjourneyman.com/blog/apple-m1-vs-intel-core-i9-deeper-look/>

Arm Cortex-X1 Core Technical Reference Manual, <https://developer.arm.com/documentation/101433/0101/Functional-description/Technical-overview/Components>

Apple CoreML Tools, <https://github.com/apple/coremltools>

Arm Website, <https://www.arm.com>

Python Bytecode Instructions, <https://docs.python.org/3/library/dis.html#python-bytecode-instructions>

Firestore Core Overview, Apple M1 Microarchitecture Research by Dougall Johnson, <https://dougallj.github.io/applecpu/firestorm.html>

Metal Performance Shaders, Apple Documentation, <https://developer.apple.com/documentation/metalperformanceshaders>

Performing Calculations on a GPU, Apple Documentation, [https://developer.apple.com/documentation/metal/performing\\_calculations\\_on\\_a\\_gpu?language=objc](https://developer.apple.com/documentation/metal/performing_calculations_on_a_gpu?language=objc)

M1 GPUs for C++ Science: Getting Started, by Lars Gebraad, <https://larsgeb.github.io/2022/04/20/m1-gpu.html>

Teardown: Identifying Apple M1's Distinct Circuit Blocks, EE Times, <https://www.eeta-sia.com/teardown-identifying-apple-m1s-distinct-circuit-blocks/>

Understanding ARM Architectures, InformIT, <https://www.informit.com/articles/article.aspx?p=1620207>

What is IP Anyway?, Eoin McCann, Arm Community Blogs, <https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/what-is-ip-anyway>

Secure Enclave, Apple Platform Security, <https://support.apple.com/guide/security/secure-enclave-sec59b0b31ff/web>

---

## NOTES

---



0  
0  
0  
0  
0  
1  
0  
1