PYTHON & PSEUDO-CODE
FOR THE
AP COMPUTER SCIENCE PRINCIPLES EXAMINATION

Python & Pseudo-Code for AP Computer Science Principles is not an AP Exam preparation book. It is a textbook for Python & Pseudo-Code, but specifically aimed at the needs of the APCS Principles student. The book contains plenty of Python programming concepts to give students enough programming knowledge for the Create Performance Task. At the same time all the programming concepts that might be tested on the End-Of-Course multiple-choice examination are thoroughly covered. The sample provided here is not a series of questions, as is typical of an AP Exam prep book, but the Table of Contents and the first 12 pages of the first chapter to provide an overview of the presentation style.

Table of Contents

Chapter I

Basic Python Features

Chapter I Topics

1.1 Introduction

This book is meant to teach high school students in the AP Computer Science Principles course an introduction to the programming language Python. It is an introduction to Python, but the material that is introduced is specifically designed to provide students with the tools necessary to complete the *Create Performance Task* of the AP Exam.

Programming is one of the seven Big Ideas presented in the APCS Principles course, but there are two cousins, *Algorithms* and *Abstraction* that are two more topics of the Big Ideas group that are very closely related to programming. These topics will be presented in detail in this book.

The teaching approach used in this course is done with a sequence of programs. Most programs are small and focus on a singular concept. The programs are meant to be looked at individually and analyzed on the computer. This is not some fiction bestseller that you can read, snuggled up inside your bed with a bunch of pillows. Ideally, you sit behind a computer and you load each one of the program examples. You can type each one of the programs, if necessary. They are quite small. Your teacher will probably provide you with the files so that the programs can be loaded quickly.

At a minimum each program must be loaded, examined, and executed. Ideally, each program should be studied and manipulated. Make changes in the program code and observe the changes in the program execution. This is called *playing with the computer*. It is really what teenagers do when they purchase a new cell phone. They check out all its features and play with the apps to learn their functionalities.

Learning to program is different from learning to use an app, but the physical playing with the computer and its functionalities are the same. It can be frustrating in the beginning, because the computer is very picky and unforgiving. A minor mistake gives you a non-functioning program. Have patience; have perseverance and you will surprise yourself. Computer Science is a very interesting field of study. It is a very fascinating and rewarding career. It is also a career where there is a great shortage of new applicants.

Learning Programming

Remember that nobody ever learned to swim by reading a book only. You need to get in a pool, get wet and start to sputter around. Pretty soon the sputtering turns into real swimming. Soon you will be competing in swim meets.

1.2 Creating Output with the print Command

You will be observing many small programs. Yes, you will learn some pretty advanced concepts, but it will be with manageable steps, one-at-a-time. In an effort to keep the focus on each program, the program code, the program execution and most of the specific explanations will be placed in one table. For each program the source code is shown first, followed by the execution output.

In this first section we will look at the **print** command. The name implies that **print** will display something, but look closely, because **print** does quite a bit more than simply provide some output. The Python programming language - like all modern programming languages - resembles English. Yes, you are lucky. You may wonder why you cannot speak or type actual English in the computer. That may come at some point in the future. Right now, the ambiguity of human languages is too great for effective communication with a computer. A computer programming language is a language that is very precise and has absolutely no exceptions in its grammar or syntax. There are no idioms. It has no statements, such as "just a second," which would be taken by a computer to be an actual second.

All programming languages have *keywords*. Each keyword has a special meaning in its language. Some of these keywords are similar to verbs in English. Verbs indicate action and our first example of an action keyword is the **print** command that is explained in the first program example. In computer science we do not say *verbs*. Instead we say *procedure* or *function* or *method.* Each program example is contained within its own table for clarity. It shows the program code, the program execution output and an explanation inside one window.

Output01.py

In the first example you see that **print** displays the literal string of characters, in this case **Hello World**, that are placed between quotes inside the parentheses. The program code shows different parts of the program with different colors. Such colors are not Python language requirements, but specific to the Integrated Development Environment (IDE), like jGRASP in this book, you use and you can change the colors. If you are looking at a printed version of this textbook, you will see black, white and shades of gray.

```
1 # Output01.py
2 # <print> displays a string constant.
3 #
4 print("Hello World")
```

```
----jGRASP exec: python Output01.py
Hello World

----jGRASP: operation complete.
```

Output02.py

The second example shows that the **print** procedure can also display a numerical literal constant. Note that quotes are not required in that case. Also observe the orange statements in lines 1, 2 and 3, which are comments. They have no impact on program execution and are strictly meant for program clarification. Comments start with the pound sign character **#**.

The line numbers in the jGRASP edit window are shown here to help identify certain statements, but they are not part of the program. The output window includes some statements created by jGRASP. The actual program output is seen between the opening and closing jGRASP statements.

```
1 # Output02.py
2 # <print> displays a numerical constant.
3 #
4 print(100)
```

```
----jGRASP exec: python Output02.py
100

----jGRASP: operation complete.
```

Output03.py

print can handle multiple outputs in one statement. In this case there are two *parameters* or *arguments* between the parentheses, separated by a comma. They are both displayed and Python inserts an automatic space during output executions. The values that are placed between the **print** parentheses - with or without quotes - are called parameters or arguments.

```
1 # Output03.py
2 # <print> displays a string and a number.
3 #
4 print("Hello World",100)
```

```
----jGRASP exec: python Output03.py
Hello World 100

----jGRASP: operation complete.
```

Output04.py

Programs with a single **print** statement hide a program execution reality. The **print** procedure not only displays the parameter information, but **print** also performs a *CRLF* at the conclusion of the program statement. *CRLF* stands for *Carriage Return, Line-Feed*. This terminology actually comes from old typewriters, which used a sliding carriage with a roller on it. Whenever, the typist reached the end of the line a handle was moved that returned the carriage to the left side and the roller turned to the next line on the paper.

```
1 # Output04.py
2 # <print> includes a CRLF.
3 #
4 print("Suzie Snodgrass")
5 print("100 Orleans Court")
6 print("Kensington, MD 20795")
```

```
 ----jGRASP exec: python Output04.py
Suzie Snodgrass
100 Orleans Court
Kensington, MD 20795

 ----jGRASP: operation complete.
```

Output05.py

Program **Output05.py** is intentionally shown without its comment heading. It also does not provide an output execution. This program includes some odd **print** statements. The **print** statements do have a set of parentheses, but they are empty. There are no arguments. What do you suppose will be the output? Think about it before you look at the program execution on your computer.

```
3 #
4 print("AAA")
5 print()
6 print("BBB")
7 print()
8 print()
9 print("CCC")
```

Output06.py

print does more than display characters and numbers. **print** also evaluates arithmetic expressions. You see that the string **"10 + 3"** is literally displayed, but with the quotes removed, Python first performs the binary arithmetic and then displays the **13** result.

```
1 # Output06.py
2 # <print> evaluates an expression and
3 # then displays the result.
4 #
5 print(10 + 3)
6 print()
7 print("10 + 3 =",10 + 3)
```

```
----jGRASP exec: python Output06.py
13

10 + 3 = 13

----jGRASP: operation complete.
```

Output07.py

Lines 4, 5 and 6 demonstrate addition, subtraction and multiplication.
Line 7 computes exponentiation, raising 10 to the 3rd power.
Line 8 and 9 perform real number division and integer number division respectively.
Line 10 performs remainder or modular division.

```
 1 # Output07.py
 2 # <print> can evaluate seven arithmetic operations.
 3 #
 4 print(10 + 3)
 5 print(10 - 3)
 6 print(10 * 3)
 7 print(10 ** 3)
 8 print(10 / 3)
 9 print(10 // 3)
10 print(10 % 3)
```

Output07.py Continued

Compare the binary operations of the previous page with the output on this page. Make changes, observe and be sure you understand each operation. MAC users, who use jGRASP with Python 3.5, may notice a different output on some programs. Please check the comment at the end of this chapter to address that issue.

```
 ----jGRASP exec: python Output07.py
13
7
30
1000
3.3333333333333335
3
1

 ----jGRASP: operation complete.
```

Output08.py

Python follows the rules of mathematical precedence. Most people remember that multiplication & division precede addition and subtraction. This example shows that exponentiation precedes other operations and parentheses come first no matter what else is going on in the mathematical expression.

Furthermore, make absolutely sure that the parentheses are correct. The number of left parentheses must always match the number of right parentheses. Python will complain with error messages when the counts do not match.

```
 1 # Output08.py
 2 # Python follows rules of precedence.
 3 #
 4 print(4 + 5 * 2)
 5 #
 6 print(4 * 5 + 2)
 7 #
 8 print((4 + 5) * 2)
 9 #
10 print(4 ** 2 * 3)
11 #
12 print(4 ** (2 * 3))
```

Output08.py Continued

```
 ----jGRASP exec: python Output08.py
14
22
18
48
4096

 ----jGRASP: operation complete.
```

Output09.py

Python has a special character. **"\n"** skips one line and **"\n\n\n"** skips three lines. This is convenient if more than one line needs skipping.

```
1 # Output09.py
2 # Using "\n" skips a line.
3 #
4 print("AAA")
5 print()
6 print("BBB\n")
7 print("CCC\n\n\n")
8 print("DDD")
```

```
 ----jGRASP exec: python Output09.py
AAA

BBB

CCC




DDD

 ----jGRASP: operation complete.
```

1.3 Using Variables in a Program

In the early days of your math courses, only constants were used. You know what they are. Constants are numbers like **5**, **13** and **127**. You added, subtracted, multiplied and divided with numbers. Later, you had more fun with fractions and decimal numbers. At some point *variables* were introduced.

In science and mathematics it is useful to express formulas and certain relationships with variables that explain some general principle. If you drive at an average rate of **60 mph** and you continue for **5 hours** at that rate, you will cover **300 miles**. On the other hand, if you drive at a rate of **45mph** for **4 hours**, you will travel **180 miles**. These two problems are examples that only use constants. The method used for computing this type of problem can be expressed in a general formula that states:

Distance = Rate × Time

The formula is normally used in its abbreviated form, which is **d = r × t**. In this formula **d**, **r** and **t** are variables. The meaning is literal. A variable is a value that is "able" to "vary" or change. A constant like **5** will always be **5**, but **d** is a variable, which changes with the values of **r** and **t**. Both **r** and **t** are also variables.

Variables make mathematics, science and computer science possible. Without variables you are very limited in the type of programs that you can write. In this section you will learn how to use variables in your programs.

A computer program is made up of words, which usually are called *keywords*. The keywords in a program have a very specific purpose, and only keywords are accepted by the Python interpreter. An interpreter will translate and execute the Python source code one line at a time; however, this only works if your program obeys all of the Python syntax rules. The first rule is that only keywords known to the Python interpreter can be used in a program.

Var01.py

In this first example three variables are used, **n1**, **n2** and **n3**. Each one of the variables is an integer variable. This is done automatically by the assignment of three integer values. Note that **print** displays the value of the variable, not its name.

This is not complicated for Python. If a string appears like **"Hello"** or **"n + 3"** then Python displays the literal string between the quotes. When there are no quotes, such as in variable **rate** then Python assumes that is a variable and its value will be used.

Var01.py Continued

```
1 # Var01.py
2 # Integer variables
3 #
4 n1 = 100
5 n2 = 200
6 n3 = 300
7 print(n1)
8 print(n2)
9 print(n3)
```

```
----jGRASP exec: python Var01.py
100
200
300

----jGRASP: operation complete.
```

Var02.py

These variable names are different from the first example. Names are not an issue to Python. The value that is assigned to each variable is a real number with a decimal point and a fraction.

```
1 # Var02.py
2 # Real number variables
3 #
4 f1 = 1.1
5 f2 = 22.22
6 f3 = 333.333
7 print(f1)
8 print(f2)
9 print(f3)
```

```
----jGRASP exec: python Var02.py
1.1
22.22
333.333

----jGRASP: operation complete.
```

Self-Documenting Variable

You can create variable names like:

boohiss = 721341.0
x = 1.385
whoknows = "Jack"

But it is far better to use self-documenting names like:

yearlySales = 721341.0
interestRate = 3.385
firstName = "Jack"

Var03.py

This program should make it clear why the quotes are so important for a program language like Python. If **howdy** has no quotes Python thinks it is a variable. This means that the statement **howdy = "howdy"** is perfectly correct in Python.

```
1  # Var03.py
2  # String variables
3  #
4  s1 = "A"
5  s2 = "John Smith"
6  s3 = "The quick brown fox jumps over the lazy dog."
7  howdy = "howdy"
8  print(s1)
9  print(s2)
10 print(s3)
11 print(howdy)
```

```
----jGRASP exec: python Var03.py
A
John Smith
The quick brown fox jumps over the lazy dog.
howdy

----jGRASP: operation complete.
```

Var04.py

Boolean variables only have two values: **True** or **False**. This adds readability to a program.

```
1  # Var04.py
2  # Boolean logic variables
3  #
4  b1 = True
5  b2 = False
6  print(b1)
7  print(b2)
```

```
----jGRASP exec: python Var04.py
True
False

----jGRASP: operation complete.
```

Var05.py

Python is perfectly happy to mix variables with literal strings. In this example the variable names are placed between quotes, which display the variable names. This is followed by the variable names used without quotes and that will display their values.

```
1  # Var05.py
2  # Output with literal strings and variables.
3  #
4  x = 100
5  y = 200
6  z = 300
7  print("x =",x)
8  print("y =",y)
9  print("z =",z)
```

```
----jGRASP exec: python Var05.py
x = 100
y = 200
z = 300

----jGRASP: operation complete.
```