BASIC I/O AND CALCULATIONS

Variables

The purpose of a variable is to keep track of a value that we need in our program. Every variable needs a name. The technical term for a variable name is an *identifier*. Python has specific rules on identifiers.

- It must begin with a letter or underscore.
- After the first character, the identifier may include letters, digits, or underscores.
- Identifiers are case sensitive.
- It may not be the name of an existing function you are using in the program (e.g. input, int, float, str, sum).
- It may not be a keyword (reserved word that already has special meaning in Python). The following table shows a list of words that you may not use for your variables. You do not need to memorize this list!

```
and, as, assert, break, class, continue, def, del, elif, else,
except, exec, False, finally, for, from, global, if, import, in,
is, lambda, None, not, or, pass, print, raise, return, True, try,
while, with, yield
```

Besides these basic rules, in the computer science community, we have a general rule of style that also advises us that our variable names should be meaningful. If you want a variable to count something, then call it something like `count`. Don't give it a generic name like `x`. Also, we generally avoid using capital letters and underscores in our variable names, unless we feel that it's necessary for a variable name to consist of more than one word.

Occasionally it is necessary for a variable name to contain more than one word. For example, suppose your program needs to keep track of the radius and mass of both a planet and its moon. In this case, a variable called `moon` or `mass` would be ambiguous, and we have to resort to using 2-word identifiers. Good variable names in this case would be:

```
planetMass
planetRadius
moonMass
moonRadius
```

Note the convention of uncapitalizing the first word and capitalizing the second word of the variable name. This is convention is called "camel case." Alternatively, you could use lowercase letters throughout, and separate the words of an identifier with an underscore, like this: `planet_mass`.

The assignment statement

This is the most important statement type. It allows us to initialize or change the value of a variable. Assignment statements appear very frequently in computer programs. This type of statement uses the = sign, which is formally called the assignment operator. In most cases, the format of an assignment statement is as follows:

< Variable > = < Expression >

To the left of the = sits the variable that we wish to define or update. To the right of the = is the value or the expression that we want to store in the variable.

An aside about my notation here: the symbols < and > when used around a word like < Variable > are called *angle brackets*. They are used to enclose the name of some category. For example, in a grammar book, you might see angle brackets used when defining the format of a sentence, like this:

< Subject > < Verb > < Direct Object >

Now, here are some examples of assignment statements:

```
rate = 1.07

name = "Mickey Mouse"

numPeople = 24

root = -b + math.sqrt(b*b - 4*a*c) / (2 * a)

area = height * (top + bottom) / 2.0
```

When we use the assignment operator =, we are issuing a command to the Python system. For example, "`rate = 1.07`" says that we want the value 1.07 to be stored in a variable called rate. This kind of statement can be translated in English as "let `rate` be 1.07" or "assign `rate` the value 1.07".

There are two possible effects of an assignment statement:

1. If the variable does not already exist, create it, and give it an initial value.
2. If the variable already exists, replace its current value with a new value.

For example, when we say "x = 5", this has two possible effects. If the variable x has never yet been used in the program up to this point, then the Python system will create a new variable x, and put the value 5 in it. But if the variable x already existed, its old value is overwritten with the value 5. The old value is then discarded. Python will not warn you if a variable in your assignment statement already exists or not. The system will assume that you know what you are doing.

The computer never forgets. When we assign a value to a variable, as in x = 5 for example, this value is stored in the variable for the rest of the program, unless we overwrite this value later. In other words, the computer will not "forget" or lose the values that you assign, until the program is finished. Upon program termination, all of its variables disappear.

For example, suppose a computer program has these statements:

```
a = 7
b = a + 1
a = 4
```

And then the program does a thousand other things. Assuming that the values of a and b are not reassigned again, at the end of the program, the values of a and b will still be 8 and 4, respectively. Just because you haven't used a variable for a long time in a program does not mean the information becomes stale or forgotten.

Discussion:

1. What values are contained in the variables a through d after the following code executes?
   ```
   a = 3
   b = 4
   c = 5
   d = c
   c = a
   b = b + a
   a = d
   ```

2. How should we respond to the following comment made by a former student?
   "Computers make no sense. For instance, they allow you to say n = n + 2. Obviously, that is a false statement. How could n possibly be equal to 2 more than itself?"

Variable types

At this point, there are three variable types that you need to be aware of. They are called `int`, `float` and `str`. They are for integers, real numbers, and strings, respectively. A variable may be of any one of these types. Better yet, you may even change the type of a variable at any time. When you introduce a variable in Python, it is not necessary to give it a type. A variable can be given or reassigned a type as the result of an assignment statement. Recall that the general format of an assignment is as follows:

< variable > = < expression >

When Python executes this statement, not only does the value of the expression get copied into the variable, but so does the type.

Therefore, Python allows you to change the type of a variable. However, for stylistic reasons, we do not recommend this. The name that you give to a variable should be a strong clue as to the type of value that you wish to store. Some quantities are naturally meant to be integer, such as a number of people. Money should be considered a real number. And text is a string. Nevertheless, the following code is legal:

```
x = "hello"
x = 2.5
x = 16
```

Here we see three consecutive assignment statements. In each case, the variable x is assigned a value, and this value has an implicit type.

Constants (e.g. “hello”, 2.5, and 16) automatically have a type. A string is easy to spot by its double quotation marks. The difference between a `float` and an `int` is that a `float` constant will always have a decimal point with digits on both sides. Thus, 4 is an `int`, while 4.0 is a `float`. Both these numbers are equal, but they are of different types. In fact, you will even sometimes see number inside a string: “4” is a string rather than an integer or real number.

Python has special built-in functions called *constructor* functions to allow us to change the type of an expression. The names of these constructor functions are simply the names of the types: int(), float(), and str(). For example:

- int(“4”) and int(4.7) both return 4
- float(4) and float(“4.0”) both return 4.0
- str(92) returns “92”

There are some simple rules concerning the type of an expression.

1. When combining expressions of the same type, the resulting type is unchanged. For example, adding integers will result in an integer.
2. When combining an integer and a float, the result will be a float. For example, 2.0 + 3 will result in 5.0, not 5.
3. The / operator performs exact division, and its result will always be a float. If you want an integer result instead (and truncate any remainder that might result), then you may use the alternative division operator //. So, remember: / means exact division, and // means truncated division. For example, 29 / 10 equals 2.9, while 29 // 10 equals 2.0.

In Python, mathematical expressions are evaluated just as in ordinary arithmetic. We follow the same order of operations:

1. Parentheses first
2. Then, exponentiation (the exponent operator is **)
3. Then, multiplication and division (operators * and / and //)
4. Finally, addition and subtraction (operators + and –)

Python has one more arithmetical operator that will be very useful for us. It is the % operator which is called modulo, or "mod" for short. The purpose of the % operator is to calculate the remainder of integer division. For example, when we do long division on integers, we compute a quotient and remainder. When we divide 14 by 3, the quotient is 4 and the remainder is 2. In Python, we would say that 14 // 3 is 4, and 14 % 3 is 2. The % operator has the same level of precedence as the multiplication and divide operators.

Besides the precedence of operators, there is also the *associativity* of operators. There are two kinds of associativity: left associativity and right associativity. Left associativity means that the evaluation proceeds from left to right. Right associativity means the evaluation goes right to left. Most operators, such as +, –, *, / and %, are left associative. For example, 8 – 3 + 2 is interpreted to mean (8 – 3) + 2. However, the ** operator is right associative. As in ordinary arithmetic, the ** is evaluated from right to left.

How to read simple input

Try these examples. Note that the syntax for reading a variable is slightly different depending on the data type that you are expecting.

To read an integer: `numPeople = int(input("How many people?  "))`

To read a real number: `price = float(input("What is the price?  "))`

To read a string of text: `city = input("What city is your destination?  ")`

Formatted output

Another essential skill is being able to print clean looking output. For example, if we are printing an amount of money, this variable is probably stored as a real number (e.g. double data type in Java or float in Python). By default, when you print a real number, you get too much precision. Formatted output is a way that you can precisely control how many digits get printed and where.

You know that to perform output in Python, you use the print() function, and often we print a string, such as `print("hello")`. If you want to perform formatted output, you also need to call the format() function inside the print() call. Here is an example:

```
number = 16
cost = 4.7
print("The cost of {0:3d} oranges is $ {1:.2f}.".format(number, cost))
```

Note the following features of the print statement:

- Inside the string that we want to print, there are place holders {0:d} and {1:.2f} that tell Python to leave room to print some value. The values are numbered starting with 0: the first value is tagged 0 and the second value is tagged as 1.
- Inside each of the placeholders, after the colon, we specify what data type is being printed. We use the letter d for integer, f for a float, and s for a string.
- When we specify the data type of a placeholder, we can optionally specify how much horizontal space to allocate to print this value. For example, "d" simply says to print an integer. But "12d" says to print the integer right justifies and leave enough space for up to 12 digits. If there are not that many digits, the unused digit positions become blank spaces. The same idea is true for printing strings. The data type code "s" means to print a string. But we can control the spacing like this: "30s" means to *left* justify the string and make room for up to 30 characters.
- Specifying space for real numbers is more interesting. We can specify both the amount of space to use to print the entire number, and we can specify the maximum number of digits after the decimal point to use. For example, if we want to print "123.45", we observe that this number requires a total of 6 symbols. And we want 2 digits after the decimal point. Therefore, the code for printing this type of number would be "6.2f". If we don't care about the total width of the number and only want to specify the precision, we can simply say ".2f".
- When we call the format() function, we specify the values that need to be printed in each of the place holders. In the above example, when we say `format(number, cost)`, we are saying that `number` is our first (0) value, and `cost` is our second (1) value.

Now we are ready to understand the meaning of the entire print statement. It says to print a line that looks something like this:

```
The cost of _____ oranges is $ _____.
```

The first blank is replaced by the value of the variable number, printed as an integer with enough room for up to a 3 digit number. The second blank is replaced by the value of the variable cost, printed to 2 decimal places. Therefore, based on the values we established, the output is:

```
The cost of  16 oranges is $ 4.70.
```

Discussion:

1. How are variables in Python different from the variables that you saw in high school algebra?

2. Show how the % operator can be used to work out the following problems:
    a. What day of the week is it 30 days from today?
    b. Is this year a leap year?
    c. What is the minimum number of pennies we need to pay 58 cents?

3. By hand, work out this expression: 30 – 3 ** 2 + 8 // 3 ** 2 * 10

4. Write a program that finds the area of a rectangle, using input given by the user.

5. What is the exact output of this code?
```
print("The square root of {0:2d} is {1:6.3f}.".format(5, 5**0.5))
```

6. What is the exact output of this code?
```
print("The {0:5s} is hungry.".format("dog"))
```

7. Suppose x and y are real numbers. We want to print x and y, and their product x*y. We want x and y to be printed to 2 decimal places, and x*y printed to 4 decimal places. What print statement should we write so that the output looks like this?
```
4.32 times 5.67 equals 24.4944
```

Shortcut assignment operators

As we saw before, = is called the assignment operator. But in Python, it is occasionally helpful to use a few more assignment operators at our disposal. They are used to make an assignment more concise. There are five of them: +=, -=, *=, /= and %=. Notice that each one consists of a certain arithmetic operator followed by an equals sign. They all work analogously. I will show you how the += works.

+= means that we want to increment (raise) the value of a variable by some amount. For example, if I want to increase the current value of x by 6, I could say x = x + 6. But using the += operator, we can simply write this as x += 6.

You are not required to use shortcut assignment operators, but it could save you from making a mistake. Consider this code: I have two variables, and I want to increase the value of each one by 1.

```
usa_grid_polygon_x = usa_grid_polygon_x + 1
usa_grid_polygon_y = usa_grid_polygon_x + 1
```

Do you see the mistake? I accidentally typed an “x” instead of a “y” near the end of the second assignment statement. Typos such as these are difficult to find. If I had used the shortcut assignment operator +=, I would not have needed to type such a long variable name twice. The correct code would look like this:

```
usa_grid_polygon_x += 1
usa_grid_polygon_y += 1
```

Errors

In computer programming, we classify errors into three categories. The word "bug" is also used as slang to refer to any of these mistakes.

Syntax error – This means that you have entered a statement that the computer does not understand. If the machine is not 100% certain of what you are trying to do, it will give you this kind of error message, and refuse to run the program. Even the most trivial typos can give rise to a syntax error.

A syntax error is usually the easiest kind of mistake to fix. The word syntax basically means grammar, so a syntax error typically means that your program contains at least one statement that violates the rules of the Python language. For example, misspelling the name of a keyword, missing or extra operator or punctuation symbol. If your program contains a syntax error, the Python system will immediately tell you about it when you try to run your program. Because the computer does not understand your program, it won't even start to run it.

Run-time error – This means that the computer understands your program, but it aborts while it's running because you are attempting to perform some operation that is impossible. In this case, the program will immediately halt during execution, and you will see an error message.

Common sources of run-time errors include trying to open a file that does not exist, dividing by zero, taking the square root of a negative number, etc. As the name suggests, a run-time error occurs while the program is running. As with syntax errors, the Python system will tell you what kind of error you have, and where in the program it thinks the error has occurred.

Logical error – This means that the program runs to completion, but the output is incorrect.

Logical errors usually result when we mistype a formula or when we print the wrong value. Logical errors are usually the most difficult kind of error to fix because the computer does not give you an error message. You must logically figure out where the error came about.

Some years ago, on the TV game show *Who Wants to be a Millionaire*, the million dollar question asked "Which insect shorted out an early supercomputer and inspired the term 'computer bug'?" The possible choices were: moth; roach; fly; Japanese beetle. The answer was a moth!

Remember: An error is simply a mistake in a program. Computer programs must be written with great care because they are brittle. A trivial change in the code can make a big difference in how the program behaves. It is very easy to make a mistake. One small change can fix or ruin a program. Don't make it a rush job. For example, when editing, it is easy to omit a statement, duplicate a statement, type statements out of order, or indent statements improperly.

Algorithm

Before we move on, I think it would be a good idea to explain one very important concept that we will need throughout the course. It is the notion of an algorithm.

An algorithm is a list of instructions written in English that explains how to solve a problem.

Does this sound familiar? How does this definition compare with the definition of a computer program that we saw earlier? They are both lists of instructions; they are both recipes for a solution. But a computer program must be written in a computer programming language, while an algorithm is written in our human language.

When we solve problems, we are basically in the business of writing algorithms. Besides being correct, an algorithm should also be:

- Unambiguous
- Detailed
- Deterministic: As we "control" the CPU, we should be clear where the next step is, and when we are finished.

Later on, we will practice creating algorithms. It's a very important skill, even more important that the nuts and bolts of any programming language.

At this point, you already know how to write some simple computer programs. From now on, we will learn a little more about Python in order to add to our arsenal. For example, you will be enlarging your knowledge about the different kind of statements that can appear in a Python program, and the different types of variables that you can use.

Some advice

The best way to learn how to solve problems is to practice. But it doesn't hurt to take a break for a few minutes and listen to some advice. ☺

To motivate a solution to a new problem, it sometimes helps to look at existing example solutions. After a while, as you accumulate some experience, you will begin to say to yourself, "This problem looks familiar." Most problems are not solved entirely from scratch. You can adapt a technique that you saw from an earlier problem. It's just like making sandwiches. If you know how to make a peanut butter and jelly sandwich, it's not hard to figure the recipe for a peanut butter and banana sandwich. And the procedure for a turkey sandwich is almost the same as for a ham sandwich.

Use built-in features of Python to simplify your solution. You can search the online documentation, and maybe you will find a very useful function. We have seen examples of this already. For example, there is already a built-in function in Python that will find the sum of the numbers in a list.

One major skill in computer problem solving is being able to read a problem description, and identifying the major "nouns" and "verbs." The nouns often become the variables in the program. The verbs could be various operations or functions.

It has been said that any skill takes about 10 years to master. In a college course you only have a few months. You won't be expected to solve every possible problem at this stage in your career. So, be patient with yourself and have some fun.

Problem solving is not easy. There is no formula that works in all cases. Difficulties can arise at any stage in the process. Ask for help if you get stuck.

It's easier to find a mistake in the (English) design of a program than in the (Python) implementation. This is why we spend so much time in step #2 in the problem-solving procedure. It's a common mistake for people to rush to step #3 and type the code before having a complete algorithm.

Input/output (I/O) is an important part of computer programs. To create an effective program, we sometimes have to use the right kind of I/O.

- Examples of input include: numbers, text, files, a URL, a button that the user can click, and image and sound
- Examples of output include: numbers, text, files, images and sound

The output is what the user sees when running your program. You need to take care that the output is in the exact specification that the user expects. For example, suppose you want to print an amount of money. An output of $3.57777777 does not look good. This situation requires *formatted* output.

<u>Suggested practice problems</u>

You may find it helpful to work with a partner for this exercise. We will discuss the solutions in class. Try your best, and don't be concerned if your first attempt at a solution is not correct.

1. The user is going to the post office, and would like to buy postage for a certain number of first-class letters and post cards. Ask the user for the number of each desired, and calculate the total cost of postage. Assume that first-class stamps are 50 cents, and post cards are 35 cents to send.

2. Use formatted output to print a line of a cash register receipt. Assume that the name of the product needs up to 15 characters, left justified, and the price can be up to 9999.99. Do not print commas.

3. Ask the user for the dimensions of a rectangular swimming pool in feet, and determine the total amount of water in gallons. Assume that there are 7.8 gallons of water per cubic foot. Print the result to one decimal place.

4. Solve a quadratic equation. Ask the user for the values of a, b, c in $ax^2 + bx + c = 0$, and determine the two answers. You may assume that the roots are real. In Python, there are two ways to calculate a square root: You can use the built-in function math.sqrt(). Or you can use the ** operator with the exponent 0.5.

5. Suppose a company compiles a report on all of its employees. The report is formatted so that there are always 4 employees listed per page. The pages are numbered sequentially, starting at 1. The employees are number sequentially, starting at 1, and appear in ascending numerical order throughout the report. Write formulas that will determine the following:
   a. If there are n employees, how many pages will be in the report?
   b. On which page will we find the report for employee number n?
   c. Page n contains the reports for which employees? In other words, give the lowest and highest numbered employee on page n.
   d. Let's generalize these formulas. How would your formulas change if we formatted the report to have P employees per page?

6. Ask the user for a car's city and highway mileage (miles per gallon), and use this information to compute the average MPG. The definition of "average" MPG assumes that 55 percent of all miles are city miles, and the remaining 45 percent are highway miles. Hint: Assume that we drive 100 miles. Therefore, we drive 55 miles in the city and 45 on the highway. The average MPG is the total number of miles driven (100) divided by the total number of gallons consumed in the city plus the highway. In your solution, you need to calculate the number of gallons consumed driving 55 miles in the city, and the number of gallons consumed driving 45 miles on the highway.