CS 161

Module 2

Exploration: Some Context

Algorithms

An algorithm is a pattern for how to manipulate and transform information. It gives a sequence of step-by-step instructions for accomplishing some task. For example, you can think of a recipe as a kind of algorithm, since it gives you a sequence of step-by-step instructions for making some food item. So why do I say that an algorithm is a pattern? Because you will usually want your algorithm to work for all cases of a problem, not just one specific case. For example, if you write a program that applies a filter to a photo, you probably want it to work for any photo, not just one specific photo. In order to write an algorithm for this task, you have to see the pattern of what's similar about applying the filter to different photos and express that pattern in your algorithm.

Computers

We take computers for granted - they're a ubiquitous part of modern society. However they're really a pretty amazing concept. Not so much dedicated computing devices that just carry out one kind of task - those are more or less an advanced form of the tool-making that people have engaged in for millennia. But the idea of a general purpose programmable machine that can carry out any computational process you describe to it (as an algorithm) was revolutionary. Not right away - Charles Babbage first described the idea for his Analytical Engine in 1837, and it wasn't until 1941, more than a century later, that the first such machine was built - but once technology and people's imaginations caught up with the idea, it transformed our world.

Most modern computers follow a similar architecture that has the following basic components:

1. A CPU (central processing unit) is the processor that decodes and executes machine language instructions. These days many computers contain multiple processors.
2. RAM (random access memory) is not actually random. "Random access" just means that you can access data anywhere in that memory. RAM is volatile, meaning that everything stored there disappears when you turn off the computer.
3. Secondary storage devices are where everything that you install or save on your computer is stored. This used to include magnetic tapes, floppy disks, and CDs, but now the most common examples are internal hard drives and USB drives, and sometimes external hard drives for backup purposes or for storing large amounts of data. This kind of memory is slower than RAM, but it doesn't go away when you turn off the computer. When you run a program or open a file, the computer will copy what you need into RAM for faster access. When you install or save something, a copy is made on a secondary storage device.

Computer Languages

I mentioned in the overview that learning to program gives you a language in which to express algorithms. Algorithms are not tied to any specific form of expression, but for a computer to understand and execute our algorithms, we must use a mathematically precise grammar and syntax, i.e. a computer language. Such a language is also useful because it forces us to be **very** specific about how we want to manipulate or transform the relevant information - there is no room for ambiguity.

Computer CPUs only understand their own **machine language**, which is pretty tedious to read or write programs in. "High-level" languages were written to be easier for humans to work with, but in order for the computer to understand them, they have to be translated to the machine language of the computer they're running on. This can be done either all at once beforehand (in a **compiled** language), or as the program is running (in an **interpreted** language). Python is an interpreted language.

Computer Programs

A computer program is just an algorithm that is expressed in some specific computer language.

Python

Python is a computer language created by Guido van Rossum, and was first released in 1991. It was named for the comedy troupe Monty Python, so in material about Python you'll occasionally run across references to their work. There are two main branches of Python - Python 2 (which is being phased out) and Python 3. They are very similar in many ways, but for this course we'll stick to Python 3.

In this course and the next one, you will not learn everything there is to know about Python - that would be impossible. You will, however, learn a lot about Python and about computer programming in general.

Interactive mode and Vocareum examples

In future lessons, you will often see interactive boxes with Python code examples for you to run and experiment with. They will look like this (this first time you'll need to agree to their use conditions, but that shouldn't happen with later ones):

You can execute the sample code by clicking on the "run" button. If you do this now, you should see "Hello world" appear in the bottom box. That is the program output. If a program asks for input, you can type that in the lower box and hit "Enter" on your keyboard. By default, when the script finishes, you'll be in Python interactive mode, so you can print out the values of variables and such.

Under "work" on the left, you can see the file(s) that are part of the example, and you can click on one to see its code. You can also modify that code and re-run it. To do that, **first exit the Python interactive mode by typing "exit()" and hitting the <enter> key**. Now you can click the "run" button again to see the results of any modifications you made. I will sometimes give instructions for things to try, but I encourage you to also try your own experiments, to satisfy your curiosity and make sure you understand the topic before moving on. If at any point you want to revert back to the original code, you can click on the "reset" button at the top right.

# Exploration: Print Statements & Types

## Print statements

The print() function can be used to print out values. You just put the value you want to print inside the parentheses. Here's an example that prints "Hello world." Click on the "Run" button to see what it does.

Try your own print statement. If you don't see the example code in the above window, click on the "+" next to "work", then click on "example.py". Change the print statement to the following:

> print(135.682)

If you ran the original code, then remember that you'll need to type in exit() and hit <enter> before running the new code.

A print statement can print more than one value - you just need to separate them with commas. By default, a space will be inserted between the values. Try this and see:

> print("pi =", 3.14159)

The two values in that print statement are different **types** - the first one is a string, the second one is a floating-point number.

We can produce the exact same output like this (give it a try!):

> print("pi = 3.14159")

In this case we're printing a single string value, not a string value and a floating-point value.

## Types

Python distinguishes between different types of values. Here we'll talk about four basic Python types: integers, floating-point numbers, strings and Booleans. We'll encounter some other types later in the course.

### **Integers**

Integers are numeric values that don't contain a decimal point. In Python 3 integers can be as long as you want (within the limits of your computer's memory).

 Floating-point numbers

Floating-point numbers are numeric values that do contain a decimal point. In mathematics, these are called real numbers. Integers are a subset of the real numbers. The value 4 is an integer, but the value 4.0 is a floating-point number.

> print(3.14159)  
> print(-12.572)  
> print(2.0)

Floats can also be written using scientific notation, using either e or E followed by a positive or negative integer (the exponent).

> print(1.2e7)  
> print(-3.27E12)  
> print(1.2e-5)

The maximum value of a float is 1.7976931348623157e+308. Floats are stored as binary fractions. Unfortunately most fractional values cannot be represented exactly in binary. Just as 1/3 cannot be represented exactly in decimal notation with a finite number of digits (0.333333333333333...), 1/10 cannot be represented exactly in binary notation with a finite number of digits (0.000110011001100...). Because of this, floats are approximate values, but the approximation is accurate to several decimal places.

### **Strings**

Strings are zero or more characters between a pair of quotation marks. You can use either single quotes or double quotes. The length of a string is only limited by your computer's memory.

If you want to have a quotation mark be part of a string, there are two ways. One way is that if the string contains a double quote, you can use single quotes to mark the ends of the string (or vice-versa).

> print('She explained, "Mae fy hofrenfad yn llawn llyswennod."')

Another way is to put a backslash character in front of the quotation marks that are part of the string.

> print("She explained, \"Mae fy hofrenfad yn llawn llyswennod.\"")

The backslash is an "escape character", because it escapes the normal significance of the following character. Because of that, if you want to have a backslash be part of a string, you need to put two of them in a row (the first one escapes the normal significance of the second one).

> print("Here is a backslash: \\")

Normal strings cannot be split across multiple lines. Later we'll see a special kind of string (called a docstring) that can be.

### **Booleans**

There are only two Boolean values - True and False. The first letter of those values must be capitalized.

> print(False)  
> print(True)

## Determining type

We can use the type() function to tell us what the type of a particular value is:

## Now you try

Now try doing the following exercises. Remember that this isn't part of your grade - it's here to reinforce your learning.

## Exercises

(See the module overview for a link to example solutions.)

1. Write a statement that prints out the float value 290.73.

Sample input: NA  
Expected Output: 290.73

2. Write a statement that prints out the string value "He replied, "Never mind", and shut the door."

Sample input: NA  
Expected Output: "He replied, "Never mind", and shut the door."

3. Write a statement that prints out the bool value True (not the string value "True").

Sample input: NA  
Expected Output: True

4. Write a statement that prints out the result of using the type command on the value 99.

Sample input: NA  
Expected Output: <class 'int'>

5. Write a statement that prints out the type of the value "What is your quest?"

Sample input: NA  
Expected Output: <class 'str'>

# Exploration: Variables & Assignment

## Variables & Assignment Statements

A variable is a name that refers to a particular value. It's called a variable because the value it refers to can change. An assignment statement assigns a value to a variable. If a variable of that name doesn't yet exist, then one is created. Assignment statements look like this:

Each of these statements causes the variable on the left side to refer to the value on the right side. It's also possible to have a variable on the right side of an assignment statement, in which case the variable on the left side will now refer to the same value as the variable on the right side. After the statement below, num\_1 and num\_2 will both refer to the value 12.

> num\_2 = num\_1

You can print the value that a variable refers to like so:

> print(radius)  
> print(first\_name)  
> print("The value of is\_even is", is\_even)

Variable names should be descriptive of their purpose to enhance the readability of your code. They must start with a letter or an underscore - subsequent characters can be numbers, letters or underscores. Variable names are case sensitive - radius, Radius, raDius, etc. would be interpreted as different variables. A variable name cannot be the same as a keyword. Python 3 has the following keywords:

False class finally is return

True continue for lambda try

None def from nonlocal while

and del global not with

as elif if or yield

assert else import pass

break except in raise

In some computer languages, you cannot change what type of value a variable refers to, but in Python you can. For example, you can assign an integer to a variable and then later assign a string to that same variable. You can use the type function to see what type of value a variable currently refers to.

### **Literals**

Literal values like 212, or -17.8, or "Wichita", or True, are often referred to simply as **literals**.

### **Constants**

If there are literal values that appear in your code, it can be a good idea to replace them with **constants**. In Python, a constant behaves just like a variable does, but by convention, the name of a constant is in all caps, for example MAX\_CAPACITY or EARTH\_GRAVITY. A constant is assigned a value once and then that value should never change during the program's execution. Python doesn't enforce this, because it has a philosophy of "we're all adults here", so you could change the value of a constant, but just don't. Declaring a constant gives a name to a literal value, making it easier to recognize or remember the value's purpose, so that you don't have "magic numbers" in your code.

Examples:

MAX\_CAPACITY = 312 # maximum number of persons that can attend an event at this venue  
EARTH\_GRAVITY = 9.81 # acceleration due to gravity at the Earth's surface, in m/s^2

### **Printing string literals filled in with the values of variables (or constants)**

Something that often comes up for the output of a program is printing out specific text, but with the values of certain variables or constants filled in. There are a few different ways of doing this in Python. Here's one example:

print("Your dog is", dog\_age, "years old.")

In this print statement the two strings are separated from the variable by commas, and each of the commas will cause a space in the output (but the spaces that aren't part of the strings will not cause spaces in the output). Here's an example that uses string concatenation, which will be mentioned in the next exploration:

print("Your dog is " + str(dog\_age) + " years old.")

In this print statement the value of dog\_age is converted to a string, and the strings are then joined with the + operator. In this version, a space was added at the end of the first string and the beginning of the last string, since there are no commas causing spaces in the output. Here's one more example, using something called an f-string (because of the 'f' at the beginning):

print(f"Your dog is {dog\_age} years old.")

In this print statement there's just a single string, and the values of any variables in curly braces are filled in.

All three of these approaches will work, so feel free to use any of them.

## Exercises

(See the module overview for a link to example solutions.)

1. Assign the value 3.14159 to a variable named pi, and then print out the value of that variable.

Sample input: NA  
Expected Output: 3.14159

2. Assign the value "Smith" to a variable named last\_name, and then print out the value of that variable.

Sample input: NA  
Expected Output: "Smith"

3. Assign the value 19.3 to a variable named length\_in\_inches, and then print out the type of the value that variable refers to.

Sample input: NA  
Expected Output: <class 'float'>

4. Assign the value "haberdashery" to a variable named occupation, and the print out the type of the value that variable refers to.

Sample input: NA  
Expected Output: <class 'str'>

Exploration: Comments, Arithmetic & String Concatenation

Comments

Comments are notes for those reading the code, and are ignored by the Python interpreter. A comment begins with #. A comment can be either on its own line or on the same line as other code. Either way, the compiler ignores everything between the # and the end of the line.

# this is a comment  
height = 1.8 # this is also a comment

You should write comments that explain non-obvious aspects of your code, such as the purpose of a particular section, the units for the value of some variable, the reason for a certain design choice, etc. You certainly don't need to comment every line, but keep in mind that how your code works may not be obvious to other people, or even to yourself in a few months.

Arithmetic

Addition, subtraction, multiplication and division are done with the +, -, \*, and / symbols.

The floor division operation is done with the // symbol, and gives you the rounded **down** result of division.

> floor\_1 = 7 // 2 # 3  
> floor\_2 = -7 // 2 # -4

**Normal division results in a float value, even if the result is an integer.  For example 8 / 4 would give you 2.0.  If you need an integer remainder, use floor division.**

The mod operation is done with the % symbol, and gives you the **remainder** of division.

> remainder\_1 = 14 % 3 # 2  
> remainder\_2 = 3 % 5 # 3

Exponentiation is done with the \*\* symbol.

> power\_1 = 3 \*\* 4 # 81  
> power\_2 = 2 \*\* -3 # 0.125

The order of operations is exponentiation, followed by multiplication, division, floor, and mod, followed by addition and subtraction. However parentheses can be used to give whatever order is needed, since operations in parentheses happen first.

> result\_1 = 3 \* 5 + 1 # 16, multiplication happens first  
> result\_2 = 3 \* (5 + 1) # 18, addition happens first

The += operator is a slightly shorter way to express that you want to add something to an existing value. The following statements do exactly the same thing:

> my\_sum = my\_sum + 8  
> my\_sum += 8

You can combine = with each of the operators above to get the following shortcut operators:

> num += 3 # same as num = num + 3  
> num -= 3 # same as num = num - 3  
> num \*= 3 # same as num = num \* 3  
> num /= 3 # same as num = num / 3  
> num //= 3 # same as num = num // 3  
> num %= 3 # same as num = num % 3  
> num \*\*= 3 # same as num = num \*\* 3

The += operator is the most commonly used of these, since it's handy for accumulating totals.

You can get the absolute value of a number like this:

answer = abs(-12)

In the above line, the variable *answer* is assigned the value 12.

Concatenation

The + operator can also be used with strings to concatenate them together.

Exercises

(See the module overview for a link to example solutions.)

1. Write a statement that prints out the remainder of 19 divided by 6.

Sample input: NA  
Expected Output: 1

2. Write a statement that prints out 2 to the 4th power.

Sample input: NA  
Expected Output: 16

3. Write a statement that finds the result of 9 divided by 5 and assigns it to a variable named *conversion\_ratio*.

Sample input: NA  
Expected Output: 1.8

4. Write code that first assigns 9 to a variable called *num\_planets*, then subtracts 1 from it, so that *num\_planets* will now equal 8.

Sample input: NA  
Expected Output: NA

5. Write code that assigns "Doctor" to a variable named *title*, assigns "Pierce" to a variable called *last\_name*, and then uses string concatenation to print out "Doctor Pierce".

Sample input: NA  
Expected Output: "Doctor Pierce"

Exploration: Input & Casting

Input Function

The input() function reads typed input from the user. You can use it in a couple of ways. For example, if you want to ask the user for their name, you can do this:

Or you can do this:

name = input("Please enter your name: ")

Notice that with the first way, your input will appear on the line after the prompt, and with the second way, your input would appear on the same line as the prompt.

User input is always a string. If you want a numeric value from user input, then you need to *cast* the string to an int or a float.

> age = int(input("Please enter your age: "))  
> height\_in\_meters = float(input("Please enter your height in meters: "))

Each of the lines above combines two functions. First the input function gets a string from the user. Next the int() or float() function casts that string to a numeric value. That value is then assigned to the variable. We can call those functions on separate lines if we want:

> str\_age = input("Please enter your age: ") # str\_age will refer to a string value  
> int\_age = int(str\_age) # int\_age will refer to an int value

If the user enters a string that cannot be cast to the specified type, that causes an error. For example, if for height in meters we try to enter "hi", what happens?

Here's an example that asks the user for two numbers and prints out their sum:

You can also cast an int to a float:

> float(18)

Or a float to an int, which truncates everything after the decimal point:

> int(3.25)

You can also cast values to strings with str().

> str(-9.4)   
> str(43)

Exercises

(See the module overview for a link to example solutions.)

1. Write code that asks the user for their name and then prints out "Hello" followed by their name.

Sample input: "Taylor Swift"  
Expected Output: "Hello Taylor Swift"

2. Write code that asks the user for two numbers and then prints out "The result is" followed by the result of multiplying those numbers.

Sample input: 3, 20  
Expected Output: 60

3. Same as #2, but first store the result of multiplication in a variable and then use that variable when printing out the result.

Sample input: 3, 20  
Expected Output: 60

# Exploration: Example Error Messages

## 

Here are a few examples of problems and their corresponding error messages:

### **EOF**

See what happens when you try to execute this code by pressing the Run button in this example.

We omitted the closing parenthesis. EOF stands for "end of file" - the interpreter got to the end of the file and never found the missing parenthesis.

### **EOL**

Here's another example. What happens if you change the code to this?

> print("hello)

We omitted the closing quotation mark for the string. EOL stands for "end of line" - the interpreter got to the end of the line without finding the missing quotation mark. Because strings can't be split across lines, the interpreter doesn't need to look all the way to the end of the file.

### **NameError**

> print(phrase)

We didn't assign a value to phrase before trying to print it.

### **SyntaxError**

> 32 = age

We have to have a variable on the left side of an assignment statement, not a literal value.

### **TypeError**

We didn't cast the input strings to numbers before trying to do math with them - the \*\* operator isn't defined for strings. Notice that if we change the operator from \*\* to +, we don't get an error message, just a result we may not expect:

### **Concatenation**

In the example above, try running the code again after changing the \*\* in line 3 to a + sign, so that the last line looks like

print("The result is", num\_1 + num\_2)

The reason this doesn't cause an error message is because + is defined for strings as concatenation. If you enter 3 for num\_1 and 4 for num\_2, then the third line will print that the result is 34, which is the concatenation of "3" and "4".

# Additional examples for Module 2

## Tinker Lab

### **Code Sample 1**

What does the following code do?

The code uses the formula P′=P(1+rn)nt. What does the **\*** operator do? What does the **\*\*** operator do?

On line 9, the **+** operator performs addition, but what does it do on line 10? What determines which way that operator will behave?

Why do we need parentheses in line 9?

The variable name num isn't very descriptive. Can you think of a better name? Try renaming that variable (you'll need to rename it everywhere that it's used).

What do you think a variable does?

What does the = operator do?

What does line 5 do? What is the significance of the # operator?

What happens if on line 6 you replace float with int? Why is int used on line 8?

Modify the code so that it calculates the amount at the end of 10 years instead of 6.

Now try changing it so that it asks for the number of years and plugs that into the formula instead of always using the same number. Constants (values that don't change) are given names in all caps, but variables are given names in lower case.

**It's possible to format the number of decimal places shown, but we won't worry about that in this course.**

### **Code sample 2**

Here's another version of the first program:

* What are two things that are done differently in this version?
* The behavior of this version is almost exactly the same, but there's a very small difference - what is it?

Note that there are usually multiple ways to accomplish the same task. Sometimes an approach is preferable for performance reasons, sometimes for style reasons, and sometimes it just comes down to personal preference. In this case, the differences fall into that third category.

### **Code sample 3**

What does the following code do?

* What do you think type means in Python?
* What's happening on lines 3, 4, 6, and 8 (that isn't happening on the other lines)?

### **Code sample 4**

What does the following code do?

* What does the **//** operator do?
* What does the **%** operator do?

Change the last line of code so that instead of having gems left over, they're given to adventurers until there are none left, so the output would be something like "3 adventurers get 7 gems and 2 adventurers get 8 gems."