Lecture 2 - Variables, Data Types and Statements

Adnan Ferdous Ashrafi

Stamford University Bangladesh

Table of Contents

What is a variable?

Definition

A variable is a name that refers to a value.

Very simply put, we use a variable when we want to store some data in our program. Every variable is connected to a value, which is the information associated with that variable.

Introducing variables to your first program

Modified Program:

```
1 message = "Hello World"
2 print(message)
3 message = "This is my first program in Python"
4 print(message)
```

Output:

```
Hello World
This is my first program in Python
```

Variables and Naming

Programmers generally choose names for their variables that are meaningful—they document what the variable is used for.

Example of Variables:

```
1 message = "And now for something completely different"
2 n = 17
3 pi = 3.14159265358979 32
```

Rules of Naming Variables

When you're using variables in Python, you need to adhere to a few rules and guidelines. Breaking some of these rules will cause errors; other guidelines just help you write code that's easier to read and understand.

- Variable names can contain only letters, numbers, and underscores. They can start with a letter or an underscore, but not with a number. For instance, you can call a variable *message_1* but not *1_message*.
- Spaces are not allowed in variable names, but underscores can be used to separate words in variable names. For example, *greeting_message* works, but *greeting message* will cause errors.

Rules of Naming Variables (Continued)

- Avoid using Python keywords and function names as variable names; that is, do not use words that Python has reserved for a particular programmatic purpose, such as the word *print*.
- Variable names should be short but descriptive. For example, *name* is better than *n*, *student_name* is better than *s_n*, and *name_length* is better than *length_of _persons_name*.
- Be careful when using the lowercase letter *l* and the uppercase letter *O* because they could be confused with the numbers 1 and 0.

Python Keywords that cannot be used in Variable Names

Python 3 has these keywords:

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

You don't have to memorize this list. In most development environments, keywords are displayed in a different color; if you try to use one as a variable name, you'll know.

What is a Assignment Statement?

Definition

An **assignment statement** creates a new variable and gives it a value.

```
1 message = "And now for something completely different"
2 n = 17
3 pi = 3.1415926535897932
```

Here the variables *message* is assigned the value "And now for something completely different". So, if we use *message* anywhere in our program then, it will display the variable's value.

Expression

An **expression** is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions:

```
>>> 42
42
>>> n
17
>>> n + 25
42
```

When you type an expression at the prompt, the interpreter evaluates it, which means that it finds the value of the expression. In this example, n has the value 17 and $n + 25$ has the value 42.

Statement

A **statement** is a unit of code that has an effect, like creating a variable or displaying a value.

```
1 >>> n = 17
2 >>> print(n)
```

The first line is an **assignment statement** that gives a value to n. The second line is a **print statement** that displays the value of n.

When you type a statement, the interpreter **executes** it, which means that it does whatever the statement says. In general, statements don't have values.

Script Mode of python

So far we have run Python in **interactive mode**, which means that you interact directly with the interpreter. Interactive mode is a good way to get started, but if you are working with more than a few lines of code, it can be clumsy.

The alternative is to save code in a file called a **script** and then run the interpreter in **script mode** to execute the script. By convention, Python scripts have names that end with *.py*.

Script Mode of python

For example, if you are using Python as a calculator, you might type

```
>>> miles = 26.2
>>> miles * 1.61
42.182
```

The first line assigns a value to miles, but it has no visible effect. The second line is an expression, so the interpreter evaluates it and displays the result. It turns out that a marathon is about 42 kilometers.

But if you type the same code into a script and run it, you get no output at all. In script mode an expression, all by itself, has no visible effect. Python evaluates the expression, but it doesn't display the result. To display the result, you need a print statement like this:

```
miles = 26.2
print(miles * 1.61)
```

Order of operation

When an expression contains more than one operator, the order of evaluation depends on the order of operations. For mathematical operators, Python follows mathematical convention. The acronym **PEMDAS** is a useful way to remember the rules:

- Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, $2 * (3 - 1)$ is 4, and $(1 + 1) * * (5 - 2)$ is 8. You can also use parentheses to make an expression easier to read, as in $(minute * 100)/60$, even if it doesn't change the result.
- Exponentiation has the next highest precedence, so $1 + 2 * * 3$ is 9, not 27, and $2 * 3 * * 2$ is 18, not 36.

Order of operation (Continued)

- Multiplication and Division have higher precedence than Addition and Subtraction. So $2*3-1$ is 5, not 4, and $6+4/2$ is 8, not 5.
- Operators with the same precedence are evaluated from left to right (except exponentiation). So in the expression $degrees/2*pi$, the division happens first and the result is multiplied by pi. To divide by 2π, you can use parentheses or write $degrees/2/pi$.

Strings

Definition

A **string** is a series of characters. Anything inside quotes is considered a string in Python, and you can use single or double quotes around your strings like this:

```
1 "This is a string."
2 'This is also a string.'
```

This flexibility allows you to use quotes and apostrophes within your strings:

```
1 'I told my friend, "Python is my favorite language!"'
2 "The language 'Python' is named after Monty Python, not the snake."
3 "One of Python's strengths is its diverse and supportive community."
```

Changing Case in a String with Methods

Definition

One of the simplest tasks you can do with strings is change the case of the words in a string.

Look at the following code, and try to determine what's happening:

```
1 name = "charlie chaplin"
2 print(name.title())
```

Output:

```
1 Charlie Chaplin
```

A method is an action that Python can perform on a piece of data. The dot (.) after name in name.title() tells Python to make the title() method act on the variable name.

Changing Case in a String with Methods (Continued)

Several other useful methods are available for dealing with case as well. For example, you can change a string to all uppercase or all lowercase letters like this:

```
1 name = "Charlie Chaplin"
2 print(name.upper())
3 print(name.lower())
```

Output:

```
1 CHARLIE CHAPLIN
2 charlie chaplin
```

Using Variables in Strings

In some situations, you'll want to use a variable's value inside a string. For example, you might want two variables to represent a first name and a last name respectively, and then want to combine those values to display someone's full name:

```
1 first_name = "charlie"
2 last_name = "chaplin"
3 full_name = f"{first_name} {last_name}"
4 print(full_name)
```

Output:

```
1 charlie chaplin
```

This f"..." is termed as **f-string** and is used in Python 3 for string formatting.

Using Variables in Strings (Continued)

You can use f-strings to compose complete messages using the information associated with a variable, as shown here:

```
1 first_name = "charlie"
2 last_name = "chaplin"
3 full_name = f"{first_name} {last_name}"
4 print(f"Hello, {full_name.title()}!")
```

Output:

```
1 Hello, Charlie Chaplin!
```

Using Variables in Strings (Continued)

You can also use f-strings to compose a message, and then assign the entire message to a variable:

```
1 first_name = "charlie"
2 last_name = "chaplin"
3 full_name = f"{first_name} {last_name}"
4 message = f"Hello, {full_name.title()}!"
5 print(message)
```

Output:

```
1 Hello, Charlie Chaplin!
```

Adding Whitespace to Strings with Tabs or Newlines

In programming, whitespace refers to any nonprinting character, such as spaces, tabs, and end-of-line symbols. You can use whitespace to organize your output so it's easier for users to read. To add a tab to your text, use the character combination \t as shown here:

```
>>> print("Python")
Python
>>> print("\tPython")
    Python
```

To add a newline in a string, use the character combination \n:

```
>>> print("Languages:\nPython\nC\nJavaScript")
Languages:
Python
C
JavaScript
```

Adding Whitespace to Strings with Tabs or Newlines

You can also combine tabs and newlines in a single string. The string "\n \t" tells Python to move to a new line, and start the next line with a tab. The following example shows how you can use a one-line string to generate four lines of output:

```
>>> print("Languages:\n\tPython\n\tC\n\tJavaScript")
Languages:
    Python
    C
    JavaScript
```

Integers

You can add (+), subtract (-), multiply (*), and divide (/) integers in Python.

```
>>> 2 + 3
5
>>> 3 - 2
1
>>> 2 * 3
6
>>> 3 / 2
1.5
```

Integers(Continued)

In a terminal session, Python simply returns the result of the operation. Python uses two multiplication symbols to represent exponents:

```
1 >>> 3 ** 2
2 9
3 >>> 3 ** 3
4 27
5 >>> 10 ** 6
6 1000000
```

Integers(Continued)

Python supports the order of operations too, so you can use multiple operations in one expression. You can also use parentheses to modify the order of operations so Python can evaluate your expression in the order you specify. For example:

```
1 >>> 2 + 3 * 4
2 14
3 >>> (2 + 3) * 4
4 20
```

Floats

Python calls any number with a decimal point a float. This term is used in most programming languages, and it refers to the fact that a decimal point can appear at any position in a number. Every programming language must be carefully designed to properly manage decimal numbers so numbers behave appropriately no matter where the decimal point appears.

```
1  >>> 0.1 + 0.1
2  0.2
3  >>> 0.2 + 0.2
4  0.4
5  >>> 2 * 0.1
6  0.2
7  >>> 2 * 0.2
8  0.4
```

Floats(Continued)

But be aware that you can sometimes get an arbitrary number of decimal places in your answer:

```
1 >>> 0.2 + 0.1
2 0.30000000000000004
3 >>> 3 * 0.1
4 0.30000000000000004
```

This happens in all languages and is of little concern. Python tries to find a way to represent the result as precisely as possible, which is sometimes difficult given how computers have to represent numbers internally.

Integers and Floats

When you divide any two numbers, even if they are integers that result in a whole number, you'll always get a float:

```
>>> 4/2
2.0
```

If you mix an integer and a float in any other operation, you'll get a float as well:

```
>>> 1 + 2.0
3.0
>>> 2 * 3.0
6.0
>>> 3.0 ** 2
9.0
```

Python defaults to a float in any operation that uses a float, even if the output is a whole number.

Underscores in Numbers

When you're writing long numbers, you can group digits using underscores to make large numbers more readable. When you print a number that was defined using underscores, Python prints only the digits:

```
>>> universe_age = 13_770_000_000
>>> print(universe_age)
13770000000
```

To Python, 1000 is the same as 1_000, which is the same as 10_00. This feature works for integers and floats, but it's only available in Python 3.6 and later.

Multiple Assignments

You can assign values to more than one variable using just a single line. For example, here's how you can initialize the variables x, y, and z to zero:

```
1 >>> x, y, z = 0, 0, 0
```

You need to separate the variable names with commas, and do the same with the values, and Python will assign each value to its respectively positioned variable. As long as the number of values matches the number of variables, Python will match them up correctly.

Constants

A *constant* is like a variable whose value stays the same throughout the life of a program. Python doesn't have built-in constant types, but Python programmers use all capital letters to indicate a variable should be treated as a constant and never be changed:

```
1 MAX_CONNECTIONS = 5000
```

When you want to treat a variable as a constant in your code, make the name of the variable all capital letters.

Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.
For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called **comments**, and they start with the # symbol:

```
1  # compute the percentage of the hour that has elapsed
2  percentage = (minute * 100) / 60
```

or

```
1  percentage = (minute * 100) / 60 # percentage of an hour
```

Everything from the # to the end of the line is ignored—it has no effect on the execution of the program.

Debugging

Three kinds of errors can occur in a program:

- **syntax** errors,
- **runtime** errors, and
- **semantic** errors

It is useful to distinguish between them in order to track them down more quickly.

Syntax Errors

Definition

"Syntax" refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so (1 + 2) is legal, but 8) is a syntax error.

If there is a syntax error anywhere in your program, Python displays an error message and quits, and you will not be able to run the program. During the first few weeks of your programming career, you might spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer errors and find them faster.

Runtime Errors

Definition

The second type of error is a runtime error, so called because the error does not appear until after the program has started running. These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened.

Runtime errors are rare in the simple programs you will see in the first few lectures, so it might be a while before you encounter one.

Semantic Errors

Definition

The third type of error is "semantic", which means related to meaning. If there is a semantic error in your program, it will run without generating error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

Further Reading

- Chapter 2 of Think Python(2nd Edition) - Allen B. Downey
- Chapter 2 of Python Crash Course - Eric Matthes
- Python Official Website
- Lecture 2 Jupyter Notebook

Thank you.
Any Questions?