CSC462 Artificial Intelligence

LAB 4: Data Types and Variables

Introduction

Python has many built-in data types such as integers, floats, booleans, strings, and lists.

By the end of this lab you will be able to:

- Explain the difference between Python's built-in data types
- Define variables with the assignment operator =
- Create variables with different data types
- Use Python's `type()` function to determine an object's data type
- Compare variables with the comparison operator ==
- Convert variables from one data type to another
- Work with integers, floats and complex numbers
- Understand the boolean data type
- Create and modify lists, dictionaries and tuples
- Index and slice strings, lists and tuples

Numeric Data Types

Python has many useful built-in *data types*. Python variables can store different types of data based on a variable's data type. A variable's data type is created dynamically, without the need to explicitly define a data type when the variable is created.

It is useful for problem solvers to understand a couple of Python's core data types in order to write well-constructed code.

A review of variable assignment in Python

Recall from the previous chapter that variables in Python are defined with the assignment operator, the equals sign =. To define a variable in Python, the variable name is written first, then the assignment operator = followed by a value or expression.

The general syntax to assign a value to variable name is below:

```
variable_name = value
```

Variable names in Python must adhere to the following rules:

- variable names must start with a letter
- variable names can only contain letters, numbers and the underscore character _
- variable names can not contain spaces or punctuation
- variable names are not enclosed in quotes or brackets

Below is a discussion of a few different built-in data types in Python.

Integers

Integers are one of the Python data types. An integer is a whole number, negative, positive or zero. In Python, integer variables are defined by assigning a whole number to a variable. Python's `type()` function can be used to determine the data type of a variable.

```
>>> a = 5
>>> type(a)
```

```
<class 'int'>
```

The output `<class 'int'>` indicates the variable `a` is an integer. Integers can be negative or zero.

```
>>> b = -2
>>> type(b)
<class 'int'>
>>> z = 0
>>> type(z)
<class 'int'>
```

Floating Point Numbers

Floating point numbers or *floats* are another Python data type. Floats are decimals, positive, negative and zero. Floats can also be represented by numbers in scientific notation which contain exponents.

Both a lower case `e` or an upper case `E` can be used to define floats in scientific notation. In Python, a float can be defined using a decimal point `.` when a variable is assigned.

```
>>> c = 6.2
>>> type(c)
<class 'float'>
>>> d = -0.03
>>> type(d)
<class 'float'>
>>> Na = 6.02e23
>>> Na
6.02e+23
>>> type(Na)
<class 'float'>
```

To define a variable as a float instead of an integer, even if the variable is assigned a whole number, a trailing decimal point `.` is used. Note the difference when a decimal point `.` comes after a whole number:

```
>>> g = 5
>>> type(g)
<class 'int'>
>>> f = 5.
>>> type(r)
<class 'float'>
```

Complex Numbers

Another useful numeric data type for problem solvers is the *complex number* data type. A complex number is defined in Python using a real component `+` an imaginary component `j`. The letter `j` must be used to denote the imaginary component. Using the letter `i` to define a complex number returns an error in Python.

```
>>> comp = 4 + 2j
>>> type(comp)
<class 'complex'>

>>> comp2 = 4 + 2i
                ^
SyntaxError: invalid syntax
```

Imaginary numbers can be added to integers and floats.

```
>>> intgr = 3
>>> type(intgr)
<class 'int'>

>>> comp_sum = comp + intgr
>>> print(comp_sum)
(7+2j)
```

```
>>> flt = 2.1
>>> comp_sum = comp + flt
>>> print(comp_sum)
(6.1+2j)
```

Boolean Data Type

The *boolean* data type is either True or False. In Python, boolean variables are defined by the `True` and `False` keywords.

```
>>> a = True
>>> type(a)
<class 'bool'>

>>> b = False
>>> type(b)
<class 'bool'>
```

The output `<class 'bool'>` indicates the variable is a boolean data type.

Note the keywords `True` and `False` must have an Upper Case first letter. Using a lowercase `true` returns an error.

```
>>> c = true
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'true' is not defined

>>> d = false
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'false' is not defined
```

Integers and Floats as Booleans

Integers and floating point numbers can be converted to the boolean data type using Python's `bool()` function. An int, float or complex number set to zero returns `False`. An integer, float or complex number set to any other number, positive or negative, returns `True`.

```
>>> zero_int = 0
>>> bool(zero_int)
False
>>> pos_int = 1
>>> bool(pos_int)
True
>>> neg_flt = -5.1
>>> bool(neg_flt)
True
```

Boolean Arithmetic

Boolean arithmetic is the arithmetic of true and false logic. A boolean or logical value can either be `True` or `False`. Boolean values can be manipulated and combined with *boolean operators*. Boolean operators in Python include `and`, `or`, and `not`.

The common boolean operators in Python are below:

- `or`
- `and`
- `not`
- == (equivalent)
- `!`= (not equivalent)

In the code section below, two variables are assigned the boolean values `True` and `False`. Then these boolean values are combined and manipulated with boolean operators.

```
>>> A = True
>>> B = False
>>> A or B
True
>>> A and B
False
>>> not A
False
>>> not B
True
>>> A == B
False
>>> A != B
True
```

Boolean operators such as `and`, `or`, and `not` can be combined with parenthesis to make compound *boolean expressions*.

```
>>> C = False
>>> A or (C and B)
True
>>> (A and B) or C
False
```

A summary of boolean arithmetic and boolean operators is shown in the table below:

A	**B**	**not A**	**not B**	**A == B**	**A =! B**	**A or B**	**A and B**
T	F	F	T	F	T	T	F
F	T	T	F	F	T	T	F
T	T	F	F	T	F	T	T
F	F	T	T	T	F	F	F

Strings

Another built-in Python data type is *strings*. Strings are sequences of letters, numbers, symbols, and spaces. In Python, strings can be almost any length and can contain spaces. Strings are assigned in Python using single quotation marks `' '` or double quotation marks `" "`.

Python strings can contain blank spaces. A blank space is a valid character in Python string.

```
>>> string = 'z'
>>>> type(string)
<class 'str'>

>>> string = 'Engineers'
>>> type(string)
<class 'str'>
```

The output `<class 'str'>` indicates the variable is a string.

Numbers as Strings

Numbers and decimals can be defined as strings too. If a decimal number is defined using quotes `' '`, the number is saved as a string rather than as a float. Integers defined using quotes become strings as well.

```
>>> num = '5.2'
>>> type(num)
<class 'str'>

>>> num = '2'
>>> type(num)
<class 'str'>
```

Strings as Boolean Values

Strings can be converted to boolean values (converted to True or False). The empty string `""` returns as `False`. All other strings convert to `True`.

```
>>> name = "Gabby"
>>> bool(name)
True
>>> empty = ""
>>> bool(empty)
False
```

Note that a string which contains just one space (`" "`) is not empty. It contains the space character. Therefore a string made up of just one space converts to `True`.

```
>>> space = " "
>>> bool(space)
True
```

String Indexing

String *indexing* is the process of pulling out specific characters from a string in a particular order. In Python, strings are indexed using square brackets `[ ]`. An important point to remember:

Python counting starts at `0` and ends at `n-1`.

Consider the word below.

```
Solution
```

The letter `S` is at index zero, the letter `o` is at index one. The last letter of the word `Solution` is `n`. `n` is in the seventh index. Even though the word `Solution` has eight letters, the last letter is in the seventh index. This is because Python indexing starts at `0` and ends at `n-1`.

Character	S	o	l	u	t	i	o	n
Index	0	1	2	3	4	5	6	7

```
>>> word = 'Solution'
>>> word[0]
'S'
>>> word[1]
'o'
>>> word[7]
'n'
```

If the eighth index of the word `Solution` is called, an error is returned.

```
>>> word[8]

IndexError: string index out of range
```

Negative Indexing

Placing a negative number inside of the square brackets pulls a character out of a string starting from the end of the string.

```
>>> word[-1]
'n'
>>> word[-2]
'o'
```

Negative Index	-8	-7	-6	-5	-4	-3	-2	-1
Character	S	o	l	u	t	i	o	n

String Slicing

String *slicing* is an operation to pull out a sequence of characters from a string. In Python, a colon on the inside of the square brackets between two numbers in a slicing operation indicates *through*. If the index `[0:3]` is called, the characters at positions `0` through `3` are returned.

Remember Python counting starts at `0` and ends at `n-1`. So `[0:3]` indicates the first through third letters, which are indexes `0` to `2`.

```
>>> word[0:3]
'Sol'
```

A colon by itself on the inside of square brackets indicates *all*.

```
>>> word[:]
'Solution'
```

When three numbers are separated by two colons inside of square brackets, the numbers represent *start* : *stop* : *step*. Remember that Python counting starts at `0` and ends at `n-1`.

```
>>> word[0:7:2]  #start:stop:step
'Slto'
```

When two colons are used inside of square brackets, and less than three numbers are specified, the missing numbers are set to their "defaults". The default start is `0`, the default stop is `n-1`, and the default step is `1`.

The two code lines below produce the same output since `0` is the default start and `7` (n-1) is the default stop. Both lines of code use a step of `2`.

```
>>> word[0:7:2]
'Slto'
>>> word[::2]
'Slto'
```

The characters that make up a string can be reversed by using the default start and stop values and specifying a step of `-1`.

```
>>> word[::-1]
'noituloS'
```

Lists

A list is a data structure in Python that can contain multiple elements of any of the other data type. A list is defined with square brackets `[ ]` and commas `,` between elements.

```
>>> lst = [ 1, 2, 3 ]
>>> type(lst)
list

>>> lst = [ 1, 5.3, '3rd_Element']
>>> type(lst)
list
```

Indexing Lists

Individual elements of a list can be accessed or *indexed* using bracket `[ ]` notation. Note that Python lists start with the index zero, not the index 1. For example:

```
>>> lst = ['statics', 'strengths', 'dynamics']
>>> lst[0]
'statics'

>>> lst[1]
'strengths'

>>> lst[2]
'dynamics'
```

Remember! Python lists start indexing at [0] not at [1]. To call the elements in a list with 3 values use: lst[0], lst[1], lst[2].

Slicing Lists

Colons `:` are used inside the square brackets to denote *all*

```
>>> lst = [2, 4, 6]
>>> lst[:]
[2, 4, 6]
```

Negative numbers can be used as indexes to call the last number of elements in the list

```
>>> lst = [2, 4, 6]
>>> lst[-1]
6
```

The colon operator can also be used to denote *all up to* and *thru end*.

```
>>> lst = [2, 4, 6]
>>> lst[:2]          # all up to 2
[2, 4]
>>> lst = [2, 4, 6]
>>> lst[2:]          # 2 thru end
[6]
```

The colon operator can also be used to denote *start : end + 1*. Note that indexing here in not inclusive. `lst[1:3]` returns the 2nd element, and 3rd element but not the fourth even though `3` is used in the index.

Remember! Python indexing is not inclusive. The last element called in an index will not be returned.

In [ ]:

Dictionaries and Tuples

Besides lists, Python has two additional data structures that can store multiple objects. These data structures are *dictionaries* and *tuples*. Tuples will be discussed first.

Tuples

Tuples are *immutable* lists. Elements of a list can be modified, but elements in a tuple can only be accessed, not modified. The name *tuple* does not mean that only two values can be stored in this data structure.

Tuples are defined in Python by enclosing elements in parenthesis `( )` and separating elements with commas. The command below creates a tuple containing the numbers `3`, `4`, and `5`.

```
>>> t_var = (3,4,5)
>>> t_var
(3, 4, 5)
```

Note how the elements of a list can be modified:

```
>>> l_var = [3,4,5]  # a list
>>> l_var[0]= 8
>>> l_var
[8, 4, 5]
```

The elements of a tuple can not be modified. If you try to assign a new value to one of the elements in a tuple, an error is returned.

```
>>> t_var = (3,4,5)  # a tuple
>>> t_var[0]= 8
>>> t_var

TypeError: 'tuple' object does not support item assignment
```

To create a tuple that just contains one numerical value, the number must be followed by a comma. Without a comma, the variable is defined as a number.

```
>>> num = (5)
>>> type(num)
int
```

When a comma is included after the number, the variable is defined as a tuple.

```
>>> t_var = (5,)
>>> type(t_var)
tuple
```

Dictionaries

Dictionaries are made up of key: value pairs. In Python, lists and tuples are organized and accessed based on position. Dictionaries in Python are organized and accessed using keys and values. The location of a pair of keys and values stored in a Python dictionary is irrelevant.

Dictionaries are defined in Python with curly braces `{ }`. Commas separate the key-value pairs that make up the dictionary. Each key-value pair is related by a colon `:`.

Let's store the ages of two people in a dictionary. The two people are `Gabby` and `Maelle`. `Gabby` is `8` and `Maelle` is `5`. Note the name `Gabby` is a string and the age `8` is an integer.

```
>>> age_dict = {"Gabby": 8 , "Maelle": 5}
>>> type(age_dict)
dict
```

The values stored in a dictionary are called and assigned using the following syntax:

```
dict_name[key] = value
>>> age_dict = {"Gabby": 8 , "Maelle": 5}
>>> age_dict["Gabby"]
8
```

We can add a new person to our `age_dict` with the following command:

```
>>> age_dict = {"Gabby": 8 , "Maelle": 5}

>>> age_dict["Peter"]= 40
>>> age_dict
{'Gabby': 8, 'Maelle': 5, 'Peter': 40}
```

Dictionaries can be converted to lists by calling the `.items()`, `.keys()`, and `.values()` methods.

```
>>> age_dict = {"Gabby": 8 , "Maelle": 5}
```

```
>>> whole_list = list(age_dict.items())
>>> whole_list
[('Gabby', 8), ('Maelle', 5)]

>>> name_list = list(age_dict.keys())
>>> name_list
['Gabby', 'Maelle']

>>> age_list = list(age_dict.values())
>>> age_list
[8, 5]
```

Items can be removed from dictionaries by calling the `.pop()` method. The dictionary key (and that key's associated value) supplied to the `.pop()` method is removed from the dictionary.

```
>>> age_dict = {"Gabby": 8 , "Maelle": 5}
>>> age_dict.pop("Gabby")
>>> age_dict
{'Maelle': 5}
```

Summary

In this lab, you learned about a couple of different data types built-in to Python. These data types include the numeric data types: integers, floats, and complex numbers. The string data type is composed of letters, numbers, spaces, and punctuation. Python also has container data types which can store many values. These container data types include lists, tuples, and dictionaries. Strings, lists and tuples can be indexed and sliced using square brackets `[ ]`.

Summary of Python Functions and Commands

Built-in Data Types

Python Data Type	**Description**
`int`	Integer
`float`	floating point number
`bool`	boolean value: True or False
`complex`	complex number, real and imaginary components
`str`	string, sequence of letters, numbers and symbols
`list`	list, formed with `[ ]`
`dict`	dictionary, formed with `{'key'=value}`
`tuple`	an immutable list, formed with `( )`

Python Functions

Function	Description
`type()`	output a variable or object data type
`len()`	return the length of a string, list dictionary or tuple
`str()`	convert a `float` or `int` into a `str` (string)
`int()`	convert a `float` or `str` into an `int` (integer)
`float()`	convert an `int` or `str` into an `float` (floating point number)

Python List Operators

Operator	Description	Example	Result
`[ ]`	indexing	`lst[1]`	`4`
`:`	start	`lst[:2]`	`[ 2, 4 ]`
`:`	end	`lst[2:]`	`[ 6, 8 ]`
`:`	through	`lst[0:3]`	`[ 2, 4, 6 ]`
`:`	start, step, end+1	`lst[0:5:2]`	`[2, 6]`