CptS 111, Spring 2023
Lect. #2, Jan. 11, 2023
Class Notes

Today's Agenda:

0. zyBook zPA 1.2.1
1. Review of `input()` and `print()` and intro to `type()`
2. What whitespace is
3. More nomenclature: expression, statement, command
4. Definitions of variables, literals, lvalues, and the assignment operator
5. Names, reserved words, and camel case
6. Floats vs integers and `int()` and `float()`

Ch. 1 (cont.)

1. `input()` , `print()` , and `type()` Functions

Last time, we ended our lecture with a few examples of input and output statements. We used the `input()` and `print()` functions. Whenever you see a command with parentheses after it, it indicates that it's either a function or a method. You can use arguments within the parentheses, but this isn't always necessary (but more on that in Ch. 5!).

Let's repeat some of what we did last time. Recall the following.

In [1]:
```python
# We prompt the user for their name.  Again, we use quotes around the
# string.  The = sign  isn't the same as it is in math.  We call it an
# assignment operator, and it assigns the results of running the input()
# command to an lvalue.

name = input('Enter your name: ')  # The = sign is the assignment operator.
                                   # It assigns the result of using the
                                   # input() function to 'name'.
```

```
Enter your name: Freddie Mercury
```

In [2]:
```python
# We can now use the name we entered because we assigned it to the lvalue
# 'name'.  'sep' is short for separator, and here the separator is no
# space, i.e., there's no space between the first string 'My name is ',
# the argument 'name', or the second string '.'
# Note the use of the commas; they're all necessary!

print('My name is ', name, '.', sep='') # add space in string, sep=''
print('My name is', name, '.', sep='')  # don't add space, sep=''
print('My name is', name, '.')          # don't add space, default sep is blank space
```

```
My name is Freddie Mercury.
My name isFreddie Mercury.
My name is Freddie Mercury .
```

Importantly, the `input()` function always returns a string! *You must remember this!*

In [3]:
```python
# Use the type() function to determine the type of an object

type(name)
```

Out[3]:
```
str
```

However, you can always use the `int()` and `float()` functions to convert a string to an integer or float, respectively.

In [4]:
```python
# Prompt user for an integer

num = input('Enter an integer: ')
```

```
Enter an integer: 42
```

In [5]:
```python
# Use the type() function to determine the type of 'num'

type(num)
```

Out[5]:
```
str
```

In [7]:
```python
# Use the int() function to convert the string returned by input() into an
# integer

num = int(num)
```

In [8]:
```python
# Voila!  We use the type() function to prove that int() did what it should

type(num)
```

Out[8]:
```
int
```

2. Whitespace

Next, let's talk a little about whitespace. We include whitespace in our code to make it more readable or to format our output. Sometimes we actually need to remove whitespace, but we'll talk about that in a much later lecture. For now, remember to:

- Use spaces between the assignment operator `=` and variable names as well as between values in an expression.
- Use `\n` or `\t` to create a newline or tab, respectively. Newlines and tabs are considered whitespace, and believe it or not, in Python they're actually characters just like 'a', 'b' and 'c'.

For example:

In [9]:
```python
# * is the multiplication operator and ** is the exponential
# operator in Python

y = (27 * 42 / 88) ** 3
```

In [10]:
```python
# Use the print() function to print the value of y.

print(y)
```

```
2139.8885119271226
```

In [11]:
```python
# Use two print commands and spaces

print('Hello,')
print('        World!')
```

```
Hello,
        World!
```

In [12]:
```python
# But better to use \n (newline) and \t (tab).

print('Hello,\n\tWorld!')
```

```
Hello,
        World!
```

In [13]:
```python
# Use \ (escape character) to print the character that follows.
# IMPORTANT, \n and \t are actually characters (text).  Here we use
# the escape character to print \n and \t.

print('Hello,\\n\\tWorld!')
```

```
Hello,\n\tWorld!
```

Ch. 2
1. More Nomenclature

- expression: an operation or operations leading to a value; an expression usually can't stand alone
- statement: a complete expression (see examples)
- command: comes from command line and is often used to describe a Python operation, e.g., the `return` command.

In [14]:
```python
# expression (not useful except as a calculator)

5 * 6 + 2 ** 40
```

Out[14]:
```
1099511627806
```

The expression above evaluates to 1099511627806, but in a computer program, it's meaningless because the value isn't stored in memory so it's unusable. Note, though, that using math expressions in Python is like using a calculator!

In [15]:
```python
# statement (or command)

y = 5 * 6 + (2 ** 40)  # Parentheses aren't needed but can add clarity
```

The statement above is complete because we have assigned the expression to the name `y`, which is usable. When a name is to the left of the assignment operator, it's called an lvalue, so here `y` is an lvalue.

In [16]:
```
# We can use y from the previous statement, but we can't use
# the previous expression because it wasn't assigned to a name.

print('y is', y)
y = y + (2 ** 99)         # Again, parentheses aren't needed
print('but y is now', y)
```

```
y is 1099511627806
but y is now 633825300114114701847863230494
```

2. Variables, Literals, Lvalues, and the Assignment Operator

All programming languages allow us to define *variables*. We call items variables when a program uses different values or changes the values of the items. OTOH, if the value of an item is fixed or constant, we call it a *literal*. A literal is usually a specific constant value, e.g., an approximation for pi or a conversion factor.

We assign values to variables (=). Recall that the symbol `=` doesn't mean 'equals' the way it does in mathematics.

Examples of variables and literals:

```
x = 5                      # 5 is an integer literal
x = x + 1                  # x is a variable
y = x * 18                 # y is a variable
y = y ** 0.5               # y is a variable (illegal in math!)
third_month = 'March'      # 'March' is a string literal
num_months_in_year = 12    # 12 is an integer literal
pi = 3.141592653           # 3.141592653 is a float literal
```

Variables and literals can be strings (which are a series of text characters; we use single or double quotes to define a string), floating point numbers, i.e., decimal values (floats), or integers. Let's look at some examples.

In [17]:
```
# String literal 'cat'

animal = 'cat'
type(animal)
```

Out[17]:
```
str
```

In [18]:
```
# Float literal 22 / 7

x = 22 / 7
type(x)
```

Out[18]:
```
float
```

In [19]:
```
# Integer literal 42

num_cars = 42
type(num_cars)
```

Out[19]:
```
int
```

The name to the left side of the assignment operator is often called an *lvalue* because it's the value on the ***left*** side. ***It's important to remember this term and what it means!*** You identify lvalues using names.

3. Names, Reserved Words, and Camel Case

Programming languages always have **rules** for naming variables, literals, and other items. In Python:

1. You can't start a name with a number.
2. Names can only use letters and numbers (with the exception of underscore _)
3. Reserved words can't be used (see Table 2.2.2 in our zyBook)
4. Uppercase and lowercase letters matter, i.e., `Num_cars` and `num_cars` are two different names.
5. You can't use a space in a name.

Different textbooks use different naming conventions. Our zyBook prefers to use underscores rather than camel case. Camel case uses uppercase letters to indicate a "word" change. For example,

```
numCars
appleWatch
blueMoon
```

We, however, will use underscores to be consistent with our textbook:

```
num_cars
apple_watch
blue_moon
```

4. Floats vs Integers, `int()` , `float()` , and `input()`

We'll refer to floating-point numbers (decimal numbers) as floats. We won't use scientific notation for floats in this course, but you should read about it in your zyBook and understand it.

A. Floats vs Integers
Whenever possible, it's better to use integers than floats when programming or to delay converting operations that will convert an integer into a float for as long as possible. This is because floats have finite precision, but integers are exact, i.e., integers are more accurate than floats. As mentioned in the first lecture, decimal integers can be converted to binary values exactly.

In [20]:
```
# A float is truncated by necessity (because memory isn't infinite)

1 / 3
```

Out[20]:
```
0.3333333333333333
```

In [21]:
```
# An integer is exact and is only limited in size by memory (RAM)

2 ** 4000
```

Out[21]:
```
131820409343094310010388979423659136318401916109327276909280345024175692811283445510797521231721220331409407564807168
230384468176942405812817310624525121840385446744443868889563289706427719939300365865529242495144888321833894158323756
200092849226089461110385787540779132654409185831255860504316472846036364908238500078268116724689002106891044880894853
471921527088201197650061259448583977618746693012787452335047965869945140544352170538037327032402834008159261693483647
994727160945768940072431686625688866030658324868306061250176433564697324072528745672177336948242366753233417556818392
219546938204560720202538843712268268448586361942128751395665874453900680147479758139717481147704392488266886671292379
541285558418744606657296304926586001793382725791100208812287673612006034789731201688939975743537276539989692230927982
557016660679726989062369216287647728379155260864643891615705346169567037448405029752790940875872989684235165316260908
983893514490200568512210790489667188789433092320719785756398772086212370409401269127676106581410793787580434036114254
547441805771508552049371634609025127325512605396392214570059772472666763440181556475095153967113514875460624794445927
79055555421362722504575706910949376
```

As we see in the two examples above, 1/3 is truncated after 16 decimal places. OTOH, 2 ** 4000 (2 to the power of 4000) is a very large number, and Python gives its exact value (the maximum value of an integer in Python depends on the amount of computer memory).

B. `input()` Function
Next let's return to the `input()` function. As mentioned before, this function *always* returns a string.

The `input()` function always returns a string.

We can use the `input()` function with or without a text prompt as its argument

We'll talk about what "return" means when we get to Ch. 5. For now, it simply means that the value produced by a function is assigned to the lvalue.

C. `int()` and `float()` Functions
Because we want to be able to input not just strings but also integers and floats and because the `input()` function always returns a string, we need a way to convert the string to an integer or float. As mentioned earlier, we do this using the `int()` or `float()` function. We illustrated this using separate statements, but we can also nest functions and convert the string returned by `input()` into an integer or float which is then assigned to the lvalue.

In [22]:
```
# input() always returns a string!

int1 = input('Enter an integer: ')
print(type(int1))
print(int1)
```

```
Enter an integer: 42
<class 'str'>
42
```

In [23]:
```
# Using two different statements to convert the input value to an integer

int2 = input('Enter an integer: ')
int2 = int(int2)
print(type(int2))
print(int2)
```

```
Enter an integer: 42
<class 'int'>
42
```

In [24]:
```python
# Using a single nested function to convert the input value to an integer

int3 = int(input('Enter an integer: '))
print(type(int3))
print(int3)
```

```
Enter an integer: 42
<class 'int'>
42
```

In [25]:
```python
# Using a single nested function to convert the input value to a float

float1 = float(input('Enter a decimal number: '))
type(float1)
```

```
Enter a decimal number: 1.618
```

Out[25]:
```
float
```

When functions are nested, the inner function is always called (executed) first, and the outermost function is called last.