Values, variables, expressions, and statements

Values

We have now seen examples of two types of values: integers and strings. As demonstrated using the **`print()`** function, these two types of values are interpreted differently. Strings, which are just sequences of characters, such as `Hello, world!`, have the designation **`str`**. Integers, which are of course whole numbers, are one type of numerical value of type **`int`**. Floating-point numbers (numbers containing a decimal point) belong to a second type called **`float`**. Numbers, both **`int`** and **`float`** type, can be treated as strings but strings cannot be treated as numbers, as we demonstrated using the **`print()`** function.

Variables

Variables can be assigned numbers (either **`int`** or **`float`**) or strings (**`str`**). For example, we can assign a sequence of As, Cs, Ts, and Gs to a variable `seq` as follows (recall that variables are assigned with the syntax `variable_name = value`):

In [1]:
```
seq = 'ATGCATGCTAC'
print(seq)
```

```
ATGCATGCTAC
```

Because we are assigning a string to the variable `dna`, as with the **`print()`** function, the value has to be in quotes. What if we were assigning a number to the variable `seq`?

In [4]:
```
seq = 5
print(seq)
seq + 5
```

```
5
```

Out[4]:
```
10
```

Things start to get a little bit tricky. If we include quotes around a number, than it becomes a string and a string no longer has a numerical value. So even though the variable `seq` may appear to be a number, it will depend on if it was assigned with or without quotes.

When assinging a number without quotes to a variable, if we include a decimal point, Python will assign **`float`** as the type by default, but if we exlude a decimal point, by deault it will be an **`int`**.

In [5]:
```
n = 5.0
```

The **`type()`** function allows us to identify the type of an object, such as a variable:

In [6]:
```
type(n)
```

Out[6]:
```
float
```

Assign a number to a variable without quotes:

In [7]:
```
n = 3
```

Use the variable in a math operation:

In [8]:
```
n*3
```

Out[8]:
```
9
```

Now try assigning a number to a variable with quotes:

In [9]:
```
n = '3'
```

Try multiplying the variable by 5 (the result may surprise you):

In [10]:
```
n*5
```

Out[10]:
```
'33333'
```

Commas are not permissible in numbers. For example, 1,000 will not mean what you expect it to:

In [11]:
```
n = 1,000
print(n)
```

```
(1, 0)
```

This is an example of a **semantic error** - valid code that doesn't do what you intended it to do. In contrast, as we've already seen many times, **sytax errors** are generated by invalid code and produce an error message.

Variable naming conventions

Variable names can be just about anything in Python but they cannot start with a number or have spaces or special characters (underscores are ok). It is best to give variables descriptive names and use lowercase letters, in particular the first letter should be lowercase (also note that variable names are case sensitive so rrna is not the same as rRNA). And although it is permissible, it is not wise to give a variable the same name as a function (such as `print`) and Python has ~30 special kewords that are off limits:

```
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
```

It is best to use variable names that are descriptive to make the code more interpretable (but be careful to avoid keywords and function names):

In [ ]:
```
x = 'ATGTGTCA' # not very informative
seq = 'ATGTGTCA' # informative at least to a biologist
seq1 = 'ATGTGTCA' # may be useful in distinguising multiple seqeuence-bas
1seq = 'ATGTGTCA' # not permissable as a variable name can't start with a
SEQ = 'ATGTGTCA' # permissable but not advisable as at least the first le
seq new = 'ATGTGTCA' # not permissable as a variable names can't contain
seq-new = 'ATGTGTCA' # not permissable as special characters are not allo
seq_new = 'ATGTGTCA' # use underscores in place of spaces
class = 'DNA' # not allowable as class is a keyword
max = 30 # permissable but not advisable as max is the name of a function
min = 10 # permissable but not advisable as min is also the name of a fun
```

Statements

In Python, a statement is any code that can be executed. `1+1` is a statement. `dna = 'ATGCC'` is also a statement. Statements can be thought of as any action or command:

In [ ]:
```
1+1
dna = 'ATGCC'
print(dna)
```

Expressions

An expression is something that represents a value or is a value. It can be a single value, a combination of values, variable, and operators, and calls to functions as long as it boils down to a single value. Expressions are things that need to be evaluated. As we saw on Tuesday, in interactive mode, the value of expressions are output upon hitting enter but in a script, they are not. Any section of code that evaluates to a value within a statment is an expression. Python programs are series of statements in a sequential order:

In [ ]:
```
num1 = 5 # Assignment statement
num2 = 7 # Assignment statement
num3 = num1 + num2 # Assignment with expression
print(num3) # Print statement
```

Arithmetic Operators

On Tuesday, we introduced the 7 arithmetic operators: `+`, `-`, `*`, `/`, `**` (to the power of), `%` (modulus), `//` (floor division).

Recall that the modulus operator yields the remainder of a division. What is the remainder of 11/3?

In [12]:
```
11%3
```

Out[12]:
```
2
```

It may not seem particularly useful now, but when working with large datasets, it can come in handy, as we'll see later on.

We saw the plus operator, `+`, in a mathematical context, but it can also be used to concatenate strings:

In [14]:
```
5+5
'color' + 'ado'
```

Out[14]:
```
'colorado'
```

The `input()` function

Python's input function, **`input()`**, allows you to collect input from a user and then perform actions on that input:

In [15]:
```
input()
```

```
ATGTA
```

Out[15]:
```
'ATGTA'
```

Not particularly useful on its own, but the input can also be stored as a variable or directly incorporated into a function.

Try storing input as a variable:

In [16]:
```
dna = input()
```

```
ATGCGT
```

Now print the user input using the **`print()`** function:

In [17]:
```
print(dna)
```

```
ATGCGT
```

It is useful to provide instructions, a prompt, when requesting input so the user knows what to do. The input function is designed to make providing a prompt very easy. Simply include the prompt within quotes within the perentheses of the input function (e.g. `input('prompt')`):

In [18]:
```
dna = input('Enter a  sequence: ')
```

```
Enter a  sequence: ATGATAT
```

By default, **`input()`** assigns whatever is input to the type **`str`**. Try prompting for a number using **`input()`**, store the number as variable and compute the inverse of the number:

In [19]:
```
num = input('Enter a number: ')
1/num
```

```
Enter a number: 5.0

---------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-19-9057e3bdc4ca> in <module>()
      1 num = input('Enter a number: ')
----> 2 1/num

TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

Fortunately, we can reassign a value to a differnt type. For example, we can convert type **`str`** to type **`int`** using **`int('string')`** (of course this will only work if the string is an integer):

In [20]:
```
num = input('Enter a number: ')
1/int(num)
```

```
Enter a number: 5
```

Out[20]:
```
0.2
```

Using `print()` with combinations of strings and variables

What if we wanted to print a literal string and the value of a variable? Python actually makes it somewhat complicated because strings require quotes and variables are not interpolated if containted in quotes.

Print the following statement: `Your sequence is variable` - where `variable` is the value assigned to the variable by the user. There are several ways to do this, for example, we can use `+` signs to delimit strings from variables. The syntax is as follows: **`print('string' + variable + 'string')`** . For example:

In [21]:
```
seq = 'ATGATA'
print('The variable is ' + seq + '.' )
```

```
The variable is ATGATA.
```

A drawback to this approach is that it does not work with **`int`** and **`float`** type values. Just as we can convert a **`str`** to an **`int`** type value if it is a whole number, we can convert an **`int`** or a **`float`** to a **`str`** . The conversion can be done by permenantly reassigning the value or simply treating as a different type within another statement:

In [29]:
```
num = 5
print('The number is ' + str(num) + '.')
```

```
The number is 5.
```

We can also use commas (**`,`**) to delimit strings and variables within a **`print()`** statement:

In [31]:
```
num = 5
print('The number is', num, '.')
```

```
The number is 5 .
```

Notice that we don't have to specify the type of **`value`** when using **`,`** . Also notice that a space is added between each value. There is yet another way that we will cover later.

We can add addiontal new lines in a **`print()`** statement using the regular expression for a new line (**`\n`**) (notice that by default the **`print()`** function adds a new line to the end of the output):

In [38]:
```
num = 5
print(num)
```

```
5
```

Exercise 2a

On the piece of paper handed out, write a script in the text editor window that prompts the user for two sequences using two separate `input()` statements, assigns the concatenated sequence to a new variable, and prints the variable within a user friendly message. Indicate all input and output in the shell window.

Test whether the code and input and output is correct on your computer.

Sneak peak: conditional statements

To evaulate if something is true or false and then perform different actions depending on the outcome, an **`if-else`** statement can be used. An **`if-else`** statement, which is a common feature of most programming languages, is a conditional statement: if some condition is true, do something; else, do something different. The else part of the statement is often optional.

Assign a number input by the user to a variable, such as `num` .

In [43]:
```
num = int(input('Enter a number: '))
```

```
Enter a number: 5
```

Now, we will use an if statement to determine if the number falls within a particular range. The Python syntax for an `if` statement is:

In [ ]:
```
if condition:
    block of code to execute
else:
    block of code to execute
```

Note the colon after the conditional statement and the indentation of code that is to be executed if the condition is met. Indentation (tab or 4 spaces) is Python's way delimiting blocks of code within conditional statements and in some other contexts that we will cover later in the course. Determine if the variable `num` is less than 10 (`<` is the Python operator for less than):

In [51]:
```
num = input('Enter a number: ')
if float(num) < 10:
    print('The number is less than 10.')
else:
    print('The number is not less than 10.')
type(num)
```

```
Enter a number: 5.0
The number is less than 10.
```

Out[51]:
```
str
```

You probably got an error message. Look closely at the error message and try to decipher the problem.

Recall that by default, the **`input()`** function treats user input as a string, even if it is a number. So we have to specifically tell Python to treat the value as a number. It doesn't matter when we do that as long as it is not after we use the value. Revise the code above to fix the problem.

What if the number has a decimal point? Recall that numbers containing decimal points belong to a different class called float so we need to specify that the value belongs to the type **`float`** .

Next week, we'll discuss comparison operators in detail, but for now, not that the **`==`** operator tests if two values are equal to one another. We'll include an else-if (**`elif`**) statement in the **`if`** statement to test if the number is equal to 10:

In [ ]:
```
if condition:
    block of code to execute
elif condition:
    block of code to execute
else:
    block of code to execute
```

In [52]:
```
num = 5
if num < 5:
    print('Number is less than 5')
elif num == 5:
    print('Number is 5')
else:
    print('Number is greater than 5')
```

```
Number is 5
```

Exercise 2b

Write a script that accepts user input in the form of a 3-nt DNA sequence (e. g. a codon) from the command line and evaluates whether or not the sequence is a start codon (ATG)? Run the script from the command line. Be sure to prompt the user to input something that can be evaulated with your script, for example, `Please enter a codon sequence:` .

Questions

Q. What are the three different types of values we've discussed so far?

A.

Q. Does Python care about white space and empty lines? If so, when?

A.

Q. When should you put quotes around the value you are assigning to a variable? Does it matter if you use single or double quotes?

A.

Q. We have now seen several functions in Python: `print()` , `input()` , `type()` , etc . What similarities do you notice between these functions? Python has many useful built-in functions listed here: https://docs.python.org/3/library/functions.html#func-str (https://docs.python.org/3/library/functions.html#func-str). Later in the course, we will discuss how to create your own functions.

A.

Additional Exercises

These exercises can be done in Jupyter Notebook, the Python interpreter, or in scripts.

1. Prompt the user for a number and print a message if the number is positive and a different message if the number is negative.
2. Modify the code in exercise 1 to also print the absolute value of the number only if it is negative. You can use the `abs()` function to get the absolute value of a number. For a description of the `abs()` function, or any other function, you can type `help(function_name)`.
3. Prompt the user for a number and identify whether the number is odd or even using only operators that we've covered so far. Hint: use the `%` operator.
4. Prompt the user for a DNA sequence and print the sequence with 5' and 3' appended to either end (e.g. user enters ATGCT, program prints 5' ATGCT 3'). Hint: use backslashes (\) to negate special characters.
5. Prompt the user for two separate sequences of DNA, concatenate them into one, and print the concatenated sequence.
6. Prompt the user for two sequences and print the concatenation of the two sequences using a single statement. You should have only one line of code.
7. Prompt the user for a DNA sequence and compute its length using the `len()` function.

Assignment 1

Assignment 1 is posted on the course website. The assignment is due this Tuesday by 10 am. Assignments must be completed in a text editor, such as TextWrangler, Notepadd++, or gedit. Submit your completed assignment on Canvas.