Introduction to Python Programming

Lets start with the obligatory hello world program:

In [ ]:
```
print("Hello World")
```

Here are some examples of defining and using variables of different types, with a few simple operators applied to the variables of different types.

Add a print statement after each line to print the type of each variable

In [3]:
```
x = True
print(type(x))
x = 3.14159
print(type(x))
x = 'True'
print(type(x))
```

```
<class 'bool'>
<class 'float'>
<class 'str'>
```

Functions

A function in Python is defined using the keyword `def`, followed by a function name, a signature within parenthises `()`, and a colon `:`. A function in python always returns a value, but if you don't specify the value to return using the **return** keyword, it returns the special value **None**.

In [8]:
```
def print_lyrics(n):
    for j in range(0, n):
        print("Who Let the Dogs out")
        for i in range(4):
            print("Who")
```

Notice when you define a function, it doesn't actually execute anything. In order to execute a function, you need to invoke it.

In [6]:
```
print_lyrics()
```

```
Who Let the Dogs out
Who
Who
Who
Who
```

In [9]:
```
print_lyrics(4)
```

```
Who Let the Dogs out
Who
Who
Who
Who
Who Let the Dogs out
Who
Who
Who
Who
Who Let the Dogs out
Who
Who
Who
Who
Who Let the Dogs out
Who
Who
Who
Who
```

Modify the previous function to take an input integer that defines how many times the entire phrase is repeated similar to the song: https://genius.com/Baha-men-who-let-the-dogs-out-lyrics

Strings

Single or double quotes designate a string literal. Python does NOT have a char datatype. Indexing into a string is accomplished via array-like syntax. Does Python use 0-indexing or 1-indexing?

In [10]:
```
a = "hello world"
print(a[1])
```

```
e
```

Substring is accomplished using square brackets:

In [11]:
```
print(a[2:7])
```

```
llo w
```

If we omit either (or both) of `start` or `stop` from `[start:stop]`, the default is the beginning and the end of the string, respectively:

In [12]:
```
a[:4]
```

Out[12]:
```
'hell'
```

We can also define the step size using the syntax `[start:end:step]` (the default value for `step` is 1, as we saw above):

In [13]:
```
a[::1]
```

Out[13]:
```
'hello world'
```

In the next cell, show what it looks like to step by a value not equal to 1.

In [15]:
```
a[::2] # step value is 2
```

Out[15]:
```
'hlowrd'
```

In the following cell, call each of the following functions on your favorite string object:

strip() removes whitespace at the beginning or the end

len() returns the length

lower() returns the string as lowercase

upper() returns the string as uppercase

replace(str1,str2) replaces string occurences of str1 with str2

split(separator) returns substrings on a string separator

In [20]:
```
b = " Hello World"
print(b.strip())
print(len(b))
print(b.lower())
print(b.upper())
print(b.replace("H", "L"))
print(b.split("-"))
```

```
Hello World
12
 hello world
 HELLO WORLD
 Lello World
[' Hello World']
```

Data Structures

Python has many powerful built in fundamental data structures.

** Lists **

Lists are used to contain a sequence of values. Lists are mutable, which means their contents can be changed after they are created, and duplicate members are also allowed. The values in a list can be of different types (which is one important way lists are fundamentally different from Java arrays). Lists are indicated syntatically in Python using the square brackets [and].

Some examples of defining lists.

In [21]:
```
cheeses = ['Cheddar', 'Edam', 'Gouda']
numbers = [17, 42, 42,'Gouda']
empty = []
print(numbers)
```

```
[17, 42, 42, 'Gouda']
```

Lists can be nested, to create complex hierarchies of information.

In [22]:
```
nested = [1, 4, [2, 8], 9, [3, 5, 7, [9, 2]]]
print(nested)
```

```
[1, 4, [2, 8], 9, [3, 5, 7, [9, 2]]]
```

Lets actually use the lists and access the values. Here we show how to access individual items, as well as multiple items known as list slices.

We can also sort lists with a simple statement .sort().

In [23]:
```
# using the cheeses list we created in a previous cell
print(cheeses[2])
print(cheeses[0])
cheeses.sort()
print(cheeses)
```

```
Gouda
Cheddar
['Cheddar', 'Edam', 'Gouda']
```

We can use negative indexes to find items relative to the end of the list.

In [24]:
```
# here again we are using the list called l, defined above in a previous cell, to access the 3rd item from the e
print(numbers[-3])
# this implies -1 always refers to the last item in the list
print(numbers[-1])
```

```
42
Gouda
```

We can use slices to access parts or multiple values of a list. Notice that the result of slicing a list is a new list.

In [25]:
```
print(numbers[1:3])
```

```
[42, 42]
```

Lists are **mutable**. We can change particular items, add/insert and remove items.

In [26]:
```
cheeses.append('Swiss')
print(cheeses)
cheeses.remove('Edam')
print(cheeses)
cheeses[1] = 'Camembert'
print(cheeses)
```

```
['Cheddar', 'Edam', 'Gouda', 'Swiss']
['Cheddar', 'Gouda', 'Swiss']
['Cheddar', 'Camembert', 'Swiss']
```

We can iterate through lists using a loop:

In [27]:
```
for i in cheeses:
    print(i)
```

```
Cheddar
Camembert
Swiss
```

Tuple

Tuples are collections that are ordered and immutable (cannot be changed). Tuples can have duplicate members.

A Tuple is defined using parentheses and can be iterated through.

In [28]:
```
pies = ("apple","blueberry","cherry","cherry")
for i in pies:
    print(i)
```

```
apple
blueberry
cherry
cherry
```

One useful convention/pattern you will run across in Python coding is returning tuples from functions. If you create a function that needs to return more than one value, you can return a tuple.

In [29]:
```
def divide(x, y):
    """Integer division of the x (dividend) by y (divisor).
    We expect x and y to be integers.
    Return both the result (quotient) and the remainder"""
    quotient = x / y
    remainder = x % y
```

```
    return (quotient, remainder)

(q, r) = divide(22, 7)
print('Result of dividing 22 / 7 is %d with a remainder of %d' % (q, r))
```

```
Result of dividing 22 / 7 is 3 with a remainder of 1
```

Notice that, since the function returns a tuple with 2 values, we need to assign the result of calling the function back into a tuple with two variables. In other words, since the `divide()` functions returns a tuple with 2 values, those 2 values are assigned into the variables `q` and `r` when returned from the function above.

The divide function above is actually a reimplementation of the built-in divmod() Python function. Notice that divmod() returns a tuple as well.

Set

Sets are collections that are unordered and and unindexed. Sets cannot have duplicate members.

A Set is defined using braces and can be iterated through.

In the following cell, write a loop to iterate through the set and add the string " pie" to each. Print out the resulting set.

In [30]:
```
pie_types = {"apple","blueberry","cherry"}
```

Dictionaries

Dictionaries are used extensively in Python programming. They are a collection, which is unordered, mutable, and indexed (using special values called keys). Dictionaries are also known as hash tables or maps in other contexts.

Like a list, a dictionary maps a key to a particular value. But lists always map an integer index/key to a particular value. A dictionary allows you to map any arbitrary key to a value. Python uses curley braces `{` and `}` syntatically to indicate dictionary data structures.

See the following example:

In [31]:
```
birthday = {'Newton': 1642,
            'Darwin': 1809,
            'Curie': 1867,
            'Einstein': 1879}
print("Newton was born in the year", birthday['Newton'])
print("Madame Curie was born in the year", birthday['Curie'])
```

```
Newton was born in the year 1642
Madame Curie was born in the year 1867
```

We can add and remove from them after being created.

Add a call at the end of the following cell to remove an item from the dictionary by making a call .pop(key), and then print out the result.

In [32]:
```
birthday['Turing'] = 1912
print(birthday)
```

```
{'Newton': 1642, 'Darwin': 1809, 'Curie': 1867, 'Einstein': 1879, 'Turing': 1912}
```

You can get a list of values in a dictionary by calling .values(). You can get the key/value pairs by calling .items().

In [33]:
```
for x in birthday.values():
    print(x)

for x,y in birthday.items():
    print(x,y)
```

```
1642
1809
1867
1879
1912
Newton 1642
Darwin 1809
Curie 1867
Einstein 1879
Turing 1912
```

In [34]:
```
help(divmod)
```

```
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.
```

Conditions and Iterations

We wouldn't be able to do much computing if we could only list a sequence of instructions that needed to be executed. Besides functions we also use conditional execution of code (**if**, **else**) and looping to execute a block of statements multiple times (**for**, **while**)

Python supports boolean **True**, and **False** as a built in data type.

In [35]:
```
print(3 == 5)
print(2 < 4)
b = True
type(b)
```

```
False
True
```

Out[35]: bool

We can build conditional execution statements using boolean or logical expressions. You should try the following cell with even and odd integer values of `x`.

For the first time, here we encounted a peculiar and unusual aspect of the Python programming language: #Program blocks are defined by their indentation level.#

Consider the Java code:

if (statement1){ System.out.println("statement1 is True\n"); } else if (statement2){ System.out.println("statement2 is True\n"); } else{ System.out.println("statement1 and statement2 are False\n"); }

In Java blocks are defined by the enclosing curly brakets { and }. And the level of indentation (white space before the code statements) does not matter (completely optional).

In Python, the extent of a code block is defined by the indentation level (usually four white spaces). This means that you must be careful to indent code correctly, or else you will get syntax errors.

In [36]:
```
x = 33
if x % 2 == 0:
    print('x is even')
else:
    print('x is odd')
```

```
x is odd
```

Change the preceeding cell to create a function that passes an integer variable into it. Call this function to test it out.

Python contains no switch/case chained conditional, instead in Python we use if, elseif, else blocks.

Here is an example. Write a function tat takes as argument an integer x and returns f(x), f being defined as:

$$f(x) = \begin{cases} \text{Not Defined} & \text{for} \quad x < 1 \\ 1 & \text{for} \quad x = 1 \\ 3 & \text{for} \quad x = 2 \\ f(x-1) \cdot f(x-2) + f(x-1) & \text{for} \quad x > 2 \end{cases}$$

In [37]:
```
def f(x):
    if x < 1:
        return None
    elif x == 1:
        return 1
    elif x == 2:
        return 3
    else:
        return f(x-1) * f(x-2) + f(x-1)

print(f(-3))
print(f(5))
```

```
None
168
```

The while statement is available for variable controlled loops, but by far the **for x in y** pattern is used much more commonly in Python in order to iterate over the elements of a data structure.

In [38]:
```
# assuming cheeses is still defined from previous cells
for c in cheeses:
    print("Here is a cheese: ", c)

# ditto, we are using the birthday dictionary from above here
```

```
for scientist, byear in birthday.items():
    print("%s was born in year %d" % (scientist, byear))
```

```
Here is a cheese:  Cheddar
Here is a cheese:  Camembert
Here is a cheese:  Swiss
Newton was born in year 1642
Darwin was born in year 1809
Curie was born in year 1867
Einstein was born in year 1879
Turing was born in year 1912
```

Acknowledgements

Original versions of these notebooks created by J.R. Johansson (robert@riken.jp) http://dml.riken.jp/~rob/ and Dr. Jay Urbain.

Modifications were made by Dr. Derek Riley