Lecture 04, 17 August 2023

Data type

- Numeric types `int` and `float`; logical values `boolean`; sequence of values `list`
- Determines what operations are allowed
  - `len(x)` does not make sense if value of `x` is not a list
- Names have no type of their own
  - Inherit their type from the values they currently hold

In [1]:
```
x = 5
```

In [2]:
```
type(x)
```

Out[2]:
```
int
```

In [3]:
```
x = False
```

In [4]:
```
type(x)
```

Out[4]:
```
bool
```

Control flow

- A Python program is a sequence of statements
  - Normal execution is sequential, top to bottom
- Most basic type of statement is **assignment**
  - `name = value`, where `value` can be an expression
- To perform interesting computations we need to control the flow
  - `if`, `for`, `while`

Functions

- Templates for re-usable code
- Instantiate with different arguments
- A function must be defined before it is used (just like any other name)
  - Typically, define your functions first, then the code that calls them

- Function definition begins with `def`
- `return(e)` gives back the answer, which can be plugged in to an expression

In [5]:
```
def square(x):
    y = x*x
    return(y)
```

In [6]:
```
square(7)
```

Out[6]:
```
49
```

- Can use value returned by a function within an expression

In [7]:
```
a = square(7) + 32 + square(71)
```

In [8]:
```
a
```

Out[8]:
```
5122
```

- Functions can use internal names, like `y` to store values local to the computation
- These names are local and cannot be accessed outside the function

In [9]:
```
y
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [9], line 1
----> 1 y

NameError: name 'y' is not defined
```

- Local names within a function do not interfere with the same names defined outside the function

In [10]:
```
y = 377
```

In [11]:
```
square(8)
```

Out[11]:
```
64
```

- Though computing `square(8)` assigns `y` to `64` inside the function, this does not affect the value of `y` outside

In [12]:
```
y
```

Out[12]:
```
377
```

- `print(e)` displays the value of its argument on the screen
- Does **not** return a value

In [13]:
```
def squaremod(x):
    y = x*x
    print(y)
```

- Invoking `squaremod()` interactively shows similar behaviour to `squaremod`

In [14]:
```
squaremod(7)
```

```
49
```

- But no value is returned, so using `squaremod()` in an expression produces an error

In [15]:
```
a = squaremod(7) + 32 + square(73)
```

```
49

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [15], line 1
----> 1 a = squaremod(7) + 32 + square(73)

TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

- `squaremod()` does not have a `return` statement - the function ends when there are no more instructions to execute
- A function that does not return a valid value implicitly returns `None`
- `None` is Python-speak for no valid value and `NoneType` is the type associated with `None`

- If we do both `print` and `return` we see the following

In [16]:
```
def squaremod2(x):
    y = x*x
    print(y)
    return(y)
```

In [17]:
```
squaremod2(9)
```

```
81
```

Out[17]:
```
81
```

- Note that the output of `return` has the label `Out[ ]` associated with it, unlike the line generated by `print`

- We can use `squaremod2` in an expression as before, but `print` will also execute along the way

In [18]:
```
a = squaremod2(7) + 32 + squaremod2(73)
```

```
49
5329
```

In [19]:
```
a
```

Out[19]:
```
5410
```

- Functions can have more than one argument

In [20]:
```
def average(a,b):
    c = (a+b)/2
    return(c)
```

- When we call the function, each parameter we pass is assigned to the corresponding argument
- In the call below, it is as though the body of the function begins with the assignments `a = 3.7` and `b = 8.55`

In [21]:
```
average(3.7,8.55)
```

Out[21]:
```
6.125
```

- It is an error to call a function with the wrong number of arguments

In [22]:
```
average(1)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [22], line 1
----> 1 average(1)

TypeError: average() missing 1 required positional argument: 'b'
```

- Here is a function with 3 arguments
- Notice that we directly evaluate an expression and return its value, without storing it in a local name

In [23]:
```
def average3(a,b,c):
    return((a+b+c)/3)
```

In [24]:
```
average3(7,9,11)
```

Out[24]:
```
9.0
```

Conditionals -- take different paths based on the values computed so far

- Basic statement is `if`

Example 1: Compute absolute value

In [25]:
```
def abs(x):
    y = x
    if x < 0:
        y = -x # Or, y = -y
    return(y)
```

In [26]:
```
abs(8), abs(-73)
```

Out[26]:
```
(8, 73)
```

- Provide an alternative to execute using `else`
- The following is equivalent to the above

In [27]:
```
def abs2(x):
    if x >= 0:
        y = x
    else:
        y = -x
    return(y)
```

- Can return from multiple places in the function.

In [28]:
```
def abs3(x):
    if x >= 0:
        return(x)
    else:
        return(-x)
```

Example 2: Check if input `x` lies in the range `[a,b]`

In [29]:
```
def inrange(x,a,b):
    if x >= a and x <= b:
        return(True)
    else:
        return(False)
```

In [30]:
```
inrange(3,2,4),inrange(2,3,4)
```

Out[30]:
```
(True, False)
```

- Shorter version, exploits the fact that what the function returns is the value of the condition tested by the `if`

In [31]:
```
def inrange2(x,a,b):
    return(x >= a and x <= b)
```

In [32]:
```
inrange2(3,2,4),inrange2(2,3,4)
```

Out[32]:
```
(True, False)
```

- Be careful when comparing floats

In [34]:
```
9 == 9.0
```

Out[34]:
```
True
```

In [35]:
```
0.1 + 0.2 - 0.3 == 0
```

Out[35]:
```
False
```

In [36]:
```
0.1 + 0.2 - 0.3
```

Out[36]:
```
5.551115123125783e-17
```

Updating lists

- Combine two lists into one - *concatenation* - `l1 + l2`
- Append a value to a list - `l.append(v)`

In [37]:
```
l1 = [1,2,3,4]
l2 = [5,6,7,8]
```

In [38]:
```
l1 + l2, l1, l2
```

Out[38]:
```
([1, 2, 3, 4, 5, 6, 7, 8], [1, 2, 3, 4], [5, 6, 7, 8])
```

In [39]:
```
l3 = l1 + l2
```

In [40]:
```
l3, l1, l2
```

Out[40]:
```
([1, 2, 3, 4, 5, 6, 7, 8], [1, 2, 3, 4], [5, 6, 7, 8])
```

- The operator `+` is overloaded to represent list concatenation as well as numeric addition
- Cannot mix types across the arguments to `+` ; error reported depends on type of first argument

In [41]:
```
l1 + 7
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [41], line 1
----> 1 l1 + 7

TypeError: can only concatenate list (not "int") to list
```

In [42]:
```
7 + l1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [42], line 1
----> 1 7 + l1

TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

- `l.append(v)` updates `l` in place, does not return a value

In [43]:
```
l3.append(9)
```

In [44]:
```
l3
```

Out[44]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [45]:
```
print(l3.append(10))
```

```
None
```

In [46]:
```
l3
```

Out[46]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Be careful not to combine in-place update with reassigning the list

In [47]:
```
l3 = l3.append(11)
```

In [48]:
```
print(l3)
```

```
None
```

- Can also append to a list using concatenation and reassignment

In [49]:
```
l3 = l1 + l2
l3 = l3 + [9]
```

In [50]:
```
l3
```

Out[50]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Example 3: Check if value `v` occurs in list `l`

In [51]:
```
def belongs(v,l):
    for x in l:
        if x == v:
            return(True)
    return(False)
```

In [52]:
```
belongs(8,l3), belongs(12,l3), belongs(8.2,l3)
```

Out[52]:
```
(True, False, False)
```

Example 4: Find the first position where `v` occurs in `l`

- If `v` is in `l`, first position lies between `0` to `len(l)-1`
- Return `-1` if no `v` in `l`

In [53]:
```
def locatepos(v,l):
    pos = 0
    for x in l:
        if x == v:
            return(pos)
        pos = pos+1
    return(-1)
```

In [54]:
```
locatepos(8,l3), locatepos(12,l3)
```

Out[54]:
```
(7, -1)
```

- We used a name `pos` to keep track of our position in the list and manually updated it with each iteration
- What we should be able to do instead is:
  - Set up a list `[0,1,2,...,len(l)-1]`
  - Run through these values and check if `l[i] == v`
  - Report the first such `i`