# Lecture 3, 27 Sep 2021 (in class)

## Names and values

### Numbers

- `int` and `float`
- Arithmetic operators
  - division `/` always produces a `float`
  - quotient `//` and remainder `%`
- Comparison operators produce value of type `bool`
  - Comparisons: `<`, `<=`, `>`, `>=`, `==`, `!=`
- Use `math` library for `log`, `sqrt`, trigonometric functions etc
  - `import math` - write `math.pi`, `math.sin(x)`, ...
  - `from math import *` - write `pi`, `sin(x)` ...
  - `import math as mt` - write `mt.pi`, `mt.sin(x)`, ...

```
# No serious limit on size of integers
x = 2**84  # Won't fit in 64 bits
y = 3**91
x*y
```

```
506470104485618930545274704870272602283938237278635903874283341873152
```

```
3/11 + 4/11
```

```
0.6363636363636364
```

### Logical values (`bool`)

- `True` and `False` are the two values of type
- Operators `not`, `or`, `and` to combine values

### Collections - lists

- A sequence of values - `[1, 3.5, True]`
- Need not be of uniform type
- length of a list - `len(l)`
- Positions are indexed `0,1,...,len(l)-1`
- Also index from right as `-1,-2,...,-len(l)`

```
mylist = [0,1,2,3,4,5,6]
```

```
mylist[6],mylist[-1]
```

```
    (6, 6)
```

## Data type

- Determines what operations are allowed
- `len(x)` does not make sense if value of `x` is not a list
- Names inherit their type from the values they currently hold
  - Not a good practice to use the same name for different types of values

## 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

```
# Statements starting with '#' are comments, not executed
x = 5
x = x+1 #·Not·a·statement·about·equality··x·<-·x+1··In·Pascal·---·x·:=·
x
```

```
    6
```

## Defining functions

```
def add3(a,b,c):  # Arguments / parameters to the function
  x = a + b + c # One or more statements to compute the value of intere
 ·return(x)     # Give the answer back
```

- colon at the end of the first line
- remaining lines are indented **uniformly**
- Be careful about spaces vs tabs

```
add3(7.5,11.3,13.9)
```

```
    32.7
```

```
def add3new(a,b,c):  # Arguments / parameters to the function
    return(a+b+c)    # can directly return an expression
```

```
add3new(7,11,13)
```

```
31
```

- A function must be defined before it is used (just like any other name)

```
add2(7,8)
def·add2(a,b):
··return(a+b)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-34-548c2f24b47f> in <module>()
----> 1 add2(7,8)
      2 def add2(a,b):
      3   return(a+b)

NameError: name 'add2' is not defined
```

SEARCH STACK OVERFLOW

```
def add2(a,b):
  return(a+b)
add2(7.5,11.3)
```

```
18.8
```

- Typically, define your functions first, then the code that calls them

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

- Basic statement is if

**Example 1**: Compute absolute value

```
def absval(x): # Returns absolute value of x
  y = x
  if y < 0:    # if boolean-expression:
     y = -y    # Indented, locally uniform, not globally uniform
  return(y)
```

```
absval(-8),absval(9)
```

```
(8, 9)
```

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

```
def absval2(x):
  if x < 0:
    v = -x
  else:
    y = x
  return(y)
```

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

```
def inrange(a,b,x):  # Return True if a <= x <= b
  if (x >= a) and (x <= b):
    return(True)
  else:
    return(False)
```

```
inrange(7,9,10)
```

```
False
```

- `return()` exits the function, so can group useful exits up front and have a final default `return()` that is the alternative to all the useful cases.

```
def inrange2(a,b,x):
  if (x >= a) and (x <= b):
    return(True)    # End of the execution if the condition holds
  return(False)     # Is reached only if condition did not hold --- same
```

```
def ineitherrange(a,b,c,d,x): # Return True if x in [a,b] or x in [c,d]
  if (x >= a) and (x <= b):
    return(True)
  if (x >= c) and (x <= d):
    return(True)
  return(False)
```

## More about lists

- concatenation of two lists: `[1,2,3] + [4,5,6]` --> `[1,2,3,4,5,6]`
- `append()` : appends a value: `l.append(4)` or `l+[4]`

```
l1 = [1,2,3]
l2 = [4,5,6]
l3 = l1+l2
```

```
l3.append(7)   # Updates l3 in place
```

```
l3 = l3 +·[8]
l3
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

```
l3 + 9
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-72-578745599419> in <module>()
----> 1 l3 + 9

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

SEARCH STACK OVERFLOW

```
9 + l3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-73-fcb39c80e8c1> in <module>()
----> 1 9 + l3

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

SEARCH STACK OVERFLOW

```
[0] + l3
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Can we extract values of different types from a list?

- Yes, the type depends on what is stored

```
l = [1,3.5,True,17.9]
x = l[0]
y = l[3]
x, type(x), y, type(y)
```

```
(1, int, 17.9, float)
```

- Whether or not you can use values of different types depends on the operations you use
- For instance, for lists `l1+l2` represents concatenation
- `add2(l1,l2)` will also work since `+` will be interpreted as list concatenation

```
add2([1,2,3],[4,5,6])
```

```
[1, 2, 3, 4, 5, 6]
```

- Can use an expression wherever a value is expected
- For instance, in the argument to `append`

```
l1 = [1,2,4]
l1.append(4+5)
l1
```

```
[1, 2, 4, 9]
```

✓ 0s completed at 4:51 PM ● ×