PDSP 2023 Lecture 03, 14 August 2023

Values and names

- Equality symbol assigns a value: `name = value`
- Names are usually referred to as *variables*

In [1]:
```
z = 82
```

In [2]:
```
z
```

Out[2]:
```
82
```

- RHS can be an expression, which can use previously assigned names

In [3]:
```
x = 38 + 53
y = x + z
z = z + 1
```

In [4]:
```
x, y, z
```

Out[4]:
```
(91, 173, 83)
```

- Using uninitialized names on RHS generates error -- first error encountered is reported
- Names are not "declared" or "announced" in advance

In [5]:
```
w = a + b
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [5], line 1
----> 1 w = a + b

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

- Not having to declare variables is not always a blessing
- Mistyping a name can raise problems that are difficult to debug

In [6]:
```
height = 1.85
heigth = 2*height
```

In [7]:
```
height
```

Out[7]:
```
1.85
```

Names, values and types

Python keeps track of names dynamically

- Values are assigned as they come
- Values have types, names inherit their type from the value

Type specifies

- Domain of values that can be used
- Set of valid operators that are available in expressions

Numeric types

Numbers can be `int` or `float`

- `int` stands for integers
- `float` stands for "floating point" = reals
  - The decimal point "floats"
  - An integer is essentially a base 2 representation of the value
  - A float is two integers: the *mantissa* with a fixed decimal point specifying the value, and the exponent telling you how much to shift the decimal point (recall scientific notation, 6.02×10^{23})
- Representation, and hence manipulation, of these types of values is different

In [8]:
```
type(7), type(3.5)
```

Out[8]:
```
(int, float)
```

In [9]:
```
height = 1.85
height= 2*height
```

In [10]:
```
type(height)
```

Out[10]:
```
float
```

Boolean type

- `True` and `False` are values of type `bool`
- Combine using `and`, `or`, `not`

Comparisons

- Comparison: m < n, a > b
- With equality: m <= n, b >= a
- Equality? Cannot be m = n, so m == n
- Not equal? m != b
- All return a value of type `bool`

In [11]:
```
x,y,z
```

Out[11]:
```
(91, 173, 83)
```

In [12]:
```
x < 100
```

Out[12]:
```
True
```

In [13]:
```
x < 100 and y < 150
```

Out[13]:
```
False
```

In [14]:
```
x < 100 or y < 150
```

Out[14]:
```
True
```

In [15]:
```
x < 100 or y > 150
```

Out[15]:
```
True
```

In [16]:
```
x < 100 and y > 150
```

Out[16]:
```
True
```

In [17]:
```
not(z > 500)
```

Out[17]:
```
True
```

Collections of values

List

- Sequence of values
- `[1,2,3,4]` is a list of integers
- Python allows mixed lists `[1,3.5,True]`

In [18]:
```
mylist = [1,3.5,True]
```

In [19]:
```
mylist
```

Out[19]:
```
[1, 3.5, True]
```

Extract an individual item from the list, by position

- Positions are numbered from 0
- `mylist = [1,3.5,True]`
- `mylist[1]` is the value at position `1`, second from start, which is `3.5`

In [20]:
```
mylist[1]
```

Out[20]:
```
3.5
```

In [21]:
```
mylist[1] = 92
```

In [22]:
```
mylist[1]
```

Out[22]:
```
92
```

In [23]:
```
mylist
```

Out[23]:
```
[1, 92, True]
```

- Illegal to access a position outside valid range
- Use `len(l)` to get the length of a list `l`

- Valid positions are `0,1,2,..., len(l)-1`

In [24]:
```
mylist[3] = False
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In [24], line 1
----> 1 mylist[3] = False

IndexError: list assignment index out of range
```

In [25]:
```
len(mylist)
```

Out[25]:
```
3
```

- Also index from right as `-1,-2,...,-len(l)`
- Note asymmetry between `0,1,...,len(l)-1` and `-1,-2,...,-len(l)`

In [26]:
```
mylist[-1]
```

Out[26]:
```
True
```

In [27]:
```
mylist[2],mylist[-2]
```

Out[27]:
```
(True, 92)
```

In [28]:
```
mylist[-3]
```

Out[28]:
```
1
```

In [29]:
```
mylist[-4]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In [29], line 1
----> 1 mylist[-4]

IndexError: list index out of range
```

Arithmetic operators

- Arithmetic: `+ , - , *`
- Division?: written `/` -- what is `7/3` ?
- Normal division / **always** produces a `float`
- "integer division" `//` which (for `int` arguments) produces an `int` = quotient
- remainder: `%`
- Given two integers `m,n, m = (m//n)*n + (m%n)`
- Exponentiation: `m ** n` (in some languages `m^n`)
- In general, if the answer requires a `float` , the type will be automatically "upgraded"

In [30]:
```
7+2.5
```

Out[30]:
```
9.5
```

In [31]:
```
7*3.0
```

Out[31]:
```
21.0
```

In [32]:
```
7/3, 7//3, 7%3
```

Out[32]:
```
(2.3333333333333335, 2, 1)
```

In [33]:
```
7.0//3.0
```

Out[33]:
```
2.0
```

In [34]:
```
2**3  # Exponentiation
# Anything after a # is ignored -- add "comments" to code
```

Out[34]:
```
8
```

Shortcuts for booleans

- A numeric value of 0 is `False`
- An empty list is `False`
- An empty string is `False`
- Anything that is not `False` is `True`

Examples

- `if n == 0 vs if not(n)`
- `if len(mylist) != 0 vs if mylist`
- `if mylist == [] vs if not(mylist)`

Standard math functions

- Need to import the `math` library

In [35]:
```
log(7), sqrt(2)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [35], line 1
----> 1 log(7), sqrt(2)

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

- Import all functions from math, qualify function names as `math.xyz()`

In [36]:
```
import math
math.log(7), math.sqrt(2)
```

Out[36]:
```
(1.9459101490553132, 1.4142135623730951)
```

- Import all functions from math without requiring qualification

In [37]:
```
from math import *
sin(pi/2), log(e)
```

Out[37]:
```
(1.0, 1.0)
```

- Import function names with qualification, but introduce an alternative (shorter) name for the library, quite commonly used practice as we shall see

In [38]:
```
import math as mt
mt.tan(mt.pi/4)
```

Out[38]:
```
0.9999999999999999
```

- Can redefine special constants imported from math library
- Check that value is restored if you reset the system

In [39]:
```
mt.pi = 3
mt.pi
```

Out[39]:
```
3
```

No serious limit on size of integers

- No special tactics to work with large magnitude values

In [40]:
```
m = 2**83
n = 3**158
p = m*n
m,n,p
```

Out[40]:
```
(9671406556917033397649408,
 242749445031546806935896183366558168250745001165697243120142488318477026168 9,
 234772857436607276377548019167468360664620245803494000255040006962648438727217432153644979136359301 12)
```

Mixing types and expressions

- Results can be unpredictable

In [41]:
```
x = 7
y = False
z = True
x and y, x and True
```

Out[41]:
```
(False, True)
```

In [42]:
```
x = 7
y = 5
x and y, not(x and y)
```

Out[42]:
```
(5, False)
```

In [44]:
```
False * x
```

Out[44]:
```
0
```

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

In [47]:
```
len(x)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [47], line 1
----> 1 len(x)

TypeError: object of type 'int' has no len()
```

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

Running Jupyter notebook

- Install Anaconda distribution
- Use `colab.research.google.com`