PDSP 2022 Lecture 03, 16 August 2022

Values and *names*

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

In [1]:
```
z = 8
```

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

Out[2]:
```
8
```

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

In [3]:
```
y = (7+3)*50
x = y*z
```

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

Out[4]:
```
(8, 500, 4000)
```

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

In [5]:
```
w = a*x
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_689045/1237844790.py in <module>
----> 1 w = a*x

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]:
```
width = 5
witdh = 10
```

In [7]:
```
width
```

Out[7]:
```
5
```

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 value with a fixed decimal point and the exponent telling you how much to shift the decimal point (recall scientific notatation, 6.02×10^{23})

- Representation, and hence manipulation, of these types of values is different

In [8]:
```
x = 5.0
y = 5
type(x), type(y)
```

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

Boolean type

- `if m < n ...`
- `True` and `False` are values of type `bool`
- Expressions are made up of `and`, `or`, `not`

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

In [12]:
```
mylist
```

Out[12]:
```
[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 [13]:
```
mylist[0],mylist[1],mylist[2]
```

Out[13]:
```
(1, 3.5, 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 [14]:
```
mylist[3]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/tmp/ipykernel_689045/920374369.py in <module>
----> 1 mylist[3]

IndexError: list index out of range
```

In [15]:
```
len(mylist)
```

Out[15]:
```
3
```

In [16]:
```
x = [5.0]
len(x)
```

Out[16]:
```
1
```

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

In [17]:
```
mylist = [0,1,2,3,4,5,6]
mylist[6],mylist[-1]
```

Out[17]:
```
(6, 6)
```

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 [18]:
```
7 // 3, 7/3, 7%3, 7.1//2.01, 8/4, 8//4, 2**3, 2**0.5
```

Out[18]:
```
(2, 2.3333333333333335, 1, 3.0, 2.0, 2, 8, 1.4142135623730951)
```

In [19]:
```
(7/3)*3
```

Out[19]:
```
7.0
```

In [ ]:
```
type(2**3), type(2.0**3), type(4**0.5)
```

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`

Mixing types and expressions

- Results can be unpredictable

In [20]:
```
y
```

Out[20]:
```
5
```

In [22]:
```
7 and y, not(7 and y)
```

Out[22]:
```
(5, False)
```

In [23]:
```
True + False
```

Out[23]:
```
1
```

Standard math functions

- Need to import the `math` library

In [25]:
```
sqrt(2), sin(pi/2)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_689045/1646495047.py in <module>
----> 1 sqrt(2), sin(pi/2)

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

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

In [26]:
```
import math
math.sqrt(2), math.sin(math.pi/2)
```

Out[26]:
```
(1.4142135623730951, 1.0)
```

- Import all functions from math without requiring qualification

In [27]:
```
from math import *
```

In [28]:
```
log(2.71818)
```

Out[28]:
```
0.9999625387017254
```

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

In [29]:
```
import math as mt
```

In [30]:
```
mt.pi
```

Out[30]:
```
3.141592653589793
```

No serious limit on size of integers

In [31]:
```
x = 2**84  # Won't fit in 64 bits
y = 3**91
x*y
```

Out[31]:
```
506470104485618930545274704870272602283938237278635903874283341873152
```

Comparisons

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

In [32]:
```
a = 7
b = 8
c = 9
```

In [34]:
```
a < 7, a < b, c >= b, a != b, b == 8
```

Out[34]:
```
(False, True, True, True, True)
```

Boolean operations

- Combine values using `and`, `or`, `not`

In [35]:
```
a < b and b < c, a > b or b < c, a > b or b > c, not(b > c)
```

Out[35]:
```
(True, True, False, True)
```

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

Functions

- Templates for re-usable code

- Instantiate with different arguments

In [7]:
```
def myadd1(a,b,c):  # Arguments / parameters to the function
  x = a + b + c # One or more statements to compute the value of interest
  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

In [8]:
```
z = 22 + myadd1(7.5,11.3,13.9)
# Implicitly assigns a to 7.5, b to 11.3, c to 13.9
z
```

Out[8]:
```
54.7
```

- If we `print` the output instead of `return`ing it, the function exits with value `None` with no type

In [4]:
```
def myadd1mod(a,b,c):  # Arguments / parameters to the function
  x = a + b + c # One or more statements to compute the value of interest
  print(x)      # Print the answer
                # Function implicitly returns when code block ends
```

In [3]:
```
z = 22 + myadd1mod(7.5,11.3,13.9)
# Implicitly assigns a to 7.5, b to 11.3, c to 13.9
z
```

```
32.7

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_744936/1396808954.py in <module>
----> 1 z = 22 + myadd1mod(7.5,11.3,13.9)
      2 # Implicitly assigns a to 7.5, b to 11.3, c to 13.9
      3 z

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

- Can directly return an expression, don't need to use an intermediate name

In [9]:
```
def myadd2(a,b,c):  # Arguments / parameters to the function
    return(a+b+c)   # Directly return an expression
```

In [10]:
```
myadd2(7,11,13)
```

Out[10]:
```
31
```

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

In [11]:
```
myadd3(7,8)
def myadd3(a,b):
  return(a+b)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_689145/412100652.py in <module>
----> 1 myadd3(7,8)
      2 def myadd3(a,b):
      3   return(a+b)

NameError: name 'myadd3' is not defined
```

In [12]:
```
def myadd3(a,b):
  return(a+b)
myadd3(7.5,11.3)
```

Out[12]:
```
18.8
```

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

Example: Find the maximum value in a list

In [23]:
```
def maxlist(l):  # Return the maximum in a list l
    if len(l) > 0:
        mymax = l[0]
        for x in l:
            if x > mymax:
                mymax = x
        return(mymax)
    else:
        print("list is empty")
        return
```

In [24]:
```
maxlist([-3,-2,-1])
```

Out[24]:
```
-1
```

In [25]:
```
z = maxlist([])
z
```

```
list is empty
```