Language features

- ***Indentation instead of braces***
- **Several sequence types**
  - Strings `'...'` : made of characters, immutable
  - Lists `[...]` : made of anything, mutable
  - *Tuples `(...)` : made of anything, immutable*
- ***Powerful subscripting (slicing)***
- ***Functions are independent entities (not all functions are methods)***
- **Exceptions as in Java**
- **Simple object system**
- **Iterators (like Java) and *generators***

A Code Sample (in IDLE)

```
x = 34 - 23              # A comment.
y = "Hello"              # Another one.
z = 3.45
if z == 3.45 or y == "Hello":
    x = x + 1
    y = y + " World"     # String concat.
print x
print y
```

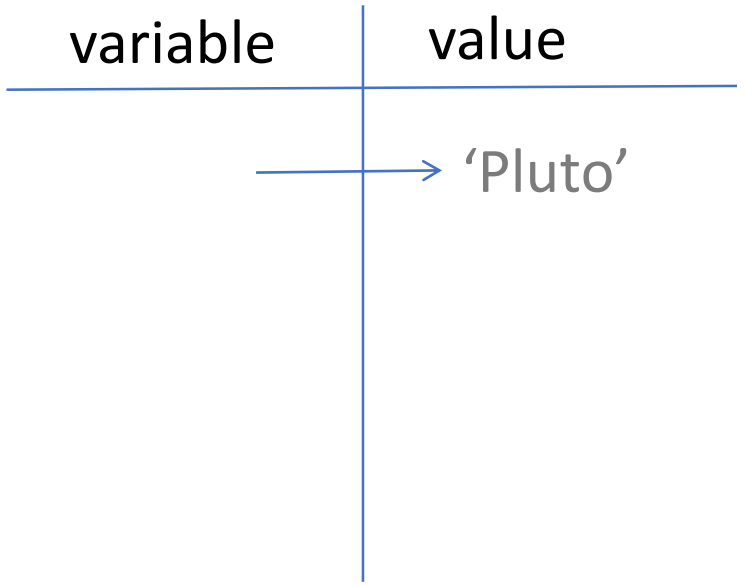
Manipulate Strings

```
>>>print ‘charles’ + ‘darwin’
charlesdarwin
>>>
```

Variables

- Variables are names for values
- Created by use – no declaration necessary

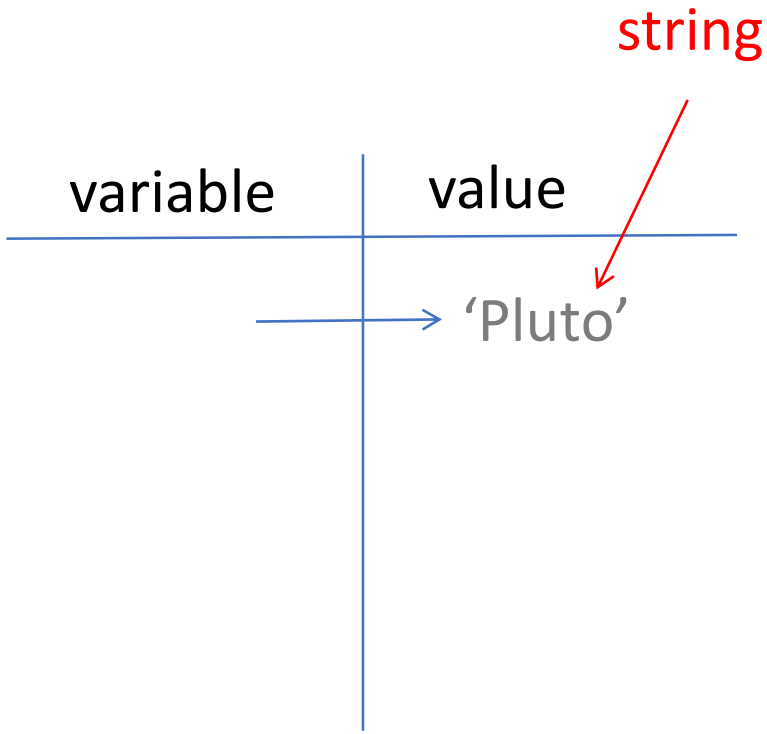
```
>>>planet = ‘Pluto’
>>>print planet
Pluto
>>>
```



Variables

- In Python, variables are just names
- Variables do not have data types

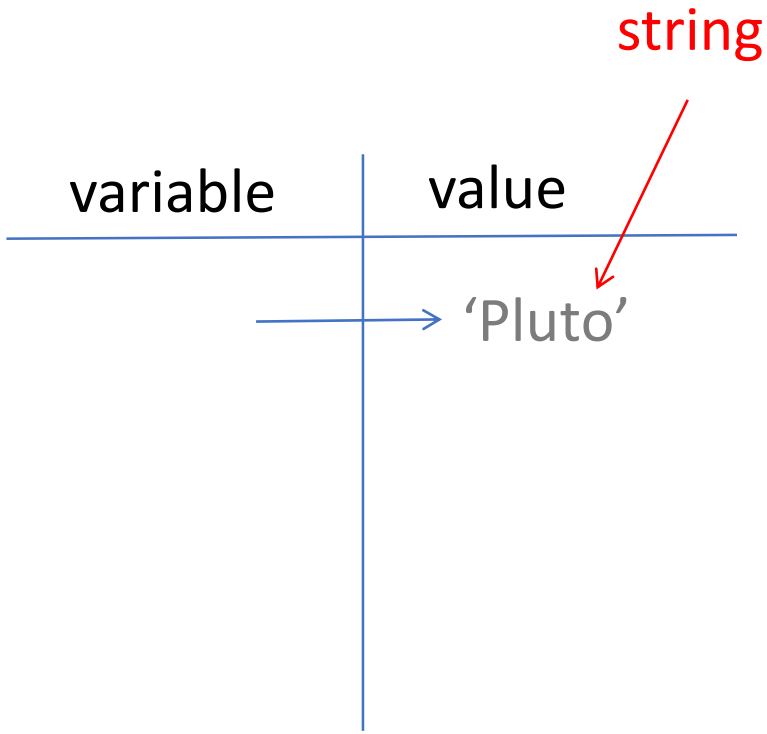
```
>>>planet = ‘Pluto’
>>>
```



Variables

- In Python, variables are just names
- Variables do not have data types

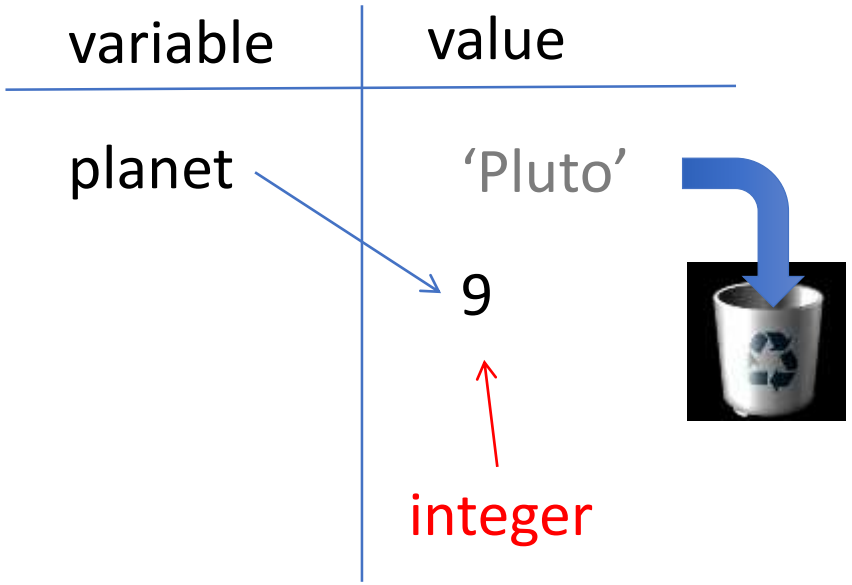
```
>>>planet = ‘Pluto’
>>>
```



Variables

- In Python, variables are just names
- Variables do not have data types

```
>>>planet = ‘Pluto’
>>>planet = 9
>>>
```



Python collects the garbage and recycles the memory (e.g., ‘Pluto’)

Variables

- You must assign a value to a variable before using it

```
>>>planet = ‘Sedna’
>>>print plant     #Note the deliberate misspelling
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    print plant
NameError: name 'plant' is not defined
```

Unlike some languages – Python does not initialize variables with a default value

Arithmetic in Python

Addition	**+**	**35 + 22**	**57**
		‘Py’ + ‘thon’	**‘Python’**
Subtraction	**-**	**35 - 22**	**13**
Multiplication	*****	**3 * 2**	**6**
		‘Py’ * 2	**‘PyPy’**
Division	**/**	**3.0 / 2**	**1.5**
		3 / 2	**1**
Exponentiation	******	**2 ** 0.5**	**1.41421356…**

Comparisons

```
>>>3 < 5
True
>>>
```

Comparisons turn numbers or strings into True or False

Comparisons

3 < 5	**True**	**Less than**
3 != 5	**True**	**Not equal to**
3 == 5	**False**	**Equal to (Notice double ==)**
3 >= 5	**False**	**Greater than or equal to**
1 < 3 < 5	**True**	**Multiple comparisons**

Single ‘=‘ is assignment
Double ‘==‘ is comparison

Values Do Have Types

```
>>>string = ‘two’
>>>number = 3
>>>print string * number
```

Values Do Have Types

```
>>>string = ‘two’
>>>number = 3
>>>print string * number  #Repeated concatenation
twotwotwo
>>>
```

Use Functions to Convert Between Types

```
>>>print int('2') + 3
5
>>>
```

Enough to Understand the Code

- **Indentation matters to the meaning of the code:**
  - Block structure indicated by indentation
- **The first assignment to a variable creates it.**
  - Variable types don't need to be declared.
  - Python figures out the variable types on its own.
- **Assignment uses `=` and comparison uses `==`.**
- **For numbers `+ - * / %` are as expected.**
  - Special use of `+` for string concatenation.
  - Special use of `%` for string formatting (as with printf in C)
- **Logical operators are words (`and, or, not`)**
  ***not* symbols**
- **Simple printing can be done with `print`.**

Basic Datatypes

- Integers (default for numbers)

```
z = 5 / 2     # Answer is 2, integer division.
```

- Floats

```
x = 3.456
```

- Strings
  - Can use “” or ‘’ to specify.
    “abc” ‘abc’ (Same thing.)
  - Unmatched can occur within the string.
    “matt’s”
  - Use triple double-quotes for multi-line strings or strings than contain both ‘ and “ inside of them:
    “““a‘b“c”””

Whitespace

Whitespace is meaningful in Python: especially indentation and placement of newlines.

- ***Use a newline to end a line of code.***
  - Use `\` when must go to next line prematurely.
- **No braces `{ }` to mark blocks of code in Python… *Use consistent indentation instead.***
  - *The first line with less indentation is outside of the block.*
  - *The first line with more indentation starts a nested block*
- **Often a colon appears at the start of a new block. (E.g. for function and class definitions.)**

Comments

- Start comments with # – the rest of line is ignored.
- Can include a "documentation string" as the first line of any new function or class that you define.
- The development environment, debugger, and other tools use it: it's good style to include one.

```
def my_function(x, y):
    """This is the docstring. This
    function does blah blah blah."""
    # The code would go here...
```

Assignment

- ***Binding a variable*** **in Python means setting a *name* to hold a *reference* to some *object*.**
  - *Assignment creates references, not copies (like Java)*
- **A variable is created *the first time* it appears on the left side of an assignment expression:**

```
x = 3
```

- **An object is deleted (by the garbage collector) once it becomes unreachable.**
- ***Names in Python do not have an intrinsic type. Objects have types.***
  - Python determines the type of the reference automatically based on what data is assigned to it.

(Multiple Assignment)

- You can also assign to multiple names at the same time.

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
```

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.

```
bob  Bob  _bob  _2_bob_  bob_2  BoB
```

- There are some reserved words:

```
and, assert, break, class, continue, def, del,
elif, else, except, exec, finally, for, from,
global, if, import, in, is, lambda, not, or, pass,
print, raise, return, try, while
```

Sequence Types

1. Tuple
   - A simple ***immutable*** ordered sequence of items
     - Immutable: a tuple cannot be modified once created....
   - Items can be of mixed types, including collection types

2. Strings
   - ***Immutable***
   - Conceptually very much like a tuple
   - Regular strings use 8-bit characters. *Unicode strings* use 2-byte characters. (All this is changed in Python 3.)

3. List
   - ***Mutable*** ordered sequence of items of mixed types

Sequence Types 2

- The three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.

- Tuples are defined using parentheses (and commas).

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Lists are defined using square brackets (and commas).

```
>>> li = ["abc", 34, 4.34, 23]
```

- Strings are defined using quotes (", ', or """).

```
>>> st = "Hello World"
>>> st = 'Hello World'
>>> st = """This is a multi-line
string that uses triple quotes."""
```

Python Lists

- A container that holds a number of other objects in a given order
- To create a list, put a number of expressions in square brackets:

```
>>> L1 = []      # This is an empty list
>>> L2 = [90,91,92]   # This list has 3 integers
>>> L3 = [‘Captain America’  ‘Iron Man’, ‘Spider Man’]
```

- Lists do not have to be homogenous

```
>>>L4 = [5, ‘Spider Man’
```

Accessing Elements in a List

- Access elements using an integer index
  item = List[index]
- List indices are zero based

```
>>> L3 = [‘Captain America’  ‘Iron Man’, ‘Spider Man’]
>>> print ‘My favorite superhero is’ + L3[2]
My favorite superhero is Spider Man
```

- To get a range of elements from a list use:

```
>>>L3[0:2]   #Get the first two items in a list
['Captain America', 'Iron Man']

     len          #Returns the number of elements in a list
3

                #Get the last item in a list
'Spider Man'
```

Sequence Types 3

- We can access individual members of a tuple, list, or string using square bracket "array" notation.
- *Note that all are 0 based…*

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]      # Second item in the tuple.
 'abc'

>>> li = ["abc", 34, 4.34, 23]
>>> li[1]       # Second item in the list.
 34

>>> st = "Hello World"
>>> st[1]    # Second character in string.
 'e'
```

Negative indices

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0.

```
>>> t[1]
'abc'
```

Negative lookup: count from right, starting with –1.

```
>>> t[-3]
4.56
```

Slicing: Return Copy of a Subset 1

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying ***before*** the second index.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

You can also use negative indices when slicing.

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

Optional argument allows selection of every n^{th} item.

```
>>> t[1:-1:2]
('abc', (2,3))
```

Slicing: Return Copy of a Subset 2

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Omit the first index to make a copy starting from the beginning of the container.

```
>>> t[:2]
(23, 'abc')
```

Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> t[2:]
(4.56, (2,3), 'def')
```

The 'in' Operator

- Boolean test whether a value is inside a collection (often called a *container* in Python:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- For strings, tests for substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- Be careful: the *in* keyword is also used in the syntax of *for loops* and *list comprehensions*.

The + Operator

- The + operator produces a *new* tuple, list, or string whose value is the concatenation of its arguments.
- Extends concatenation from strings to other types

```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)

>>> [1, 2, 3] + [4, 5, 6]
 [1, 2, 3, 4, 5, 6]

>>> "Hello" + " " + "World"
 'Hello World'
```

Lists: Mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
   ['abc', 45, 4.34, 23]
```

- We can change lists *in place.*
- Name *li* still points to the same memory reference when we're done.

Tuples: Immutable

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14

Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item assignment
```

You can't change a tuple.

You can make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```

- *The immutability of tuples means they're faster than lists.*

Tuples: Immutable

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14

Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item assignment
```

You can't change a tuple.

You can make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```

- *The immutability of tuples means they're faster than lists.*

Operations on Lists Only 1

```
>>> li = [1, 11, 3, 4, 5]

>>> li.append('a')  # Note the method syntax
>>> li
[1, 11, 3, 4, 5, 'a']

>>> li.insert(2, 'i')
>>>li
[1, 11, 'i', 3, 4, 5, 'a']
```

Operations on Lists Only 3

```
>>> li = ['a', 'b', 'c', 'b']

>>> li.index('b')      # index of first occurrence*
1
```

*more complex forms exist

```
>>> li.count('b')      # number of occurrences
2

>>> li.remove('b')     # remove first occurrence
>>> li
  ['a', 'c', 'b']
```

Operations on Lists Only 4

```
>>> li = [5, 2, 6, 8]

>>> li.reverse()     # reverse the list *in place*
>>> li
  [8, 6, 2, 5]

>>> li.sort()        # sort the list *in place*
>>> li
  [2, 5, 6, 8]

>>> li.sort(some_function)
    # sort in place using user-defined comparison
```

Tuples vs. Lists

- **Lists slower but more powerful than tuples.**
  - Lists can be modified, and they have lots of handy operations we can perform on them.
  - Tuples are immutable and have fewer features.

- **To convert between tuples and lists use the list() and tuple() functions:**

```
li = list(tu)
tu = tuple(li)
```

Creating and accessing dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user']
'bozo'

>>> d['pswd']
1234

>>> d['bozo']

Traceback (innermost last):
  File '<interactive input>' line 1, in ?
KeyError: bozo
```

Creating and accessing dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user']
'bozo'

>>> d['pswd']
1234

>>> d['bozo']

Traceback (innermost last):
  File '<interactive input>' line 1, in ?
KeyError: bozo
```

Updating Dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user'] = 'clown'
>>> d
{'user':'clown', 'pswd':1234}
```

- Keys must be unique.
- Assigning to an existing key replaces its value.

```
>>> d['id'] = 45
>>> d
{'user':'clown', 'id':45, 'pswd':1234}
```

- Dictionaries are unordered
  - New entry might appear anywhere in the output.
- (Dictionaries work by *hashing*)

Removing dictionary entries

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> del d['user']              # Remove one. Note that del is
                               # a function.
>>> d
{'p':1234, 'i':34}

>>> d.clear()                  # Remove all.
>>> d
{}
>>> a=[1,2]
>>> del a[1]                  # (del also works on lists)
>>> a
[1]
```

Useful Accessor Methods

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> d.keys()                # List of current keys
['user', 'p', 'i']

>>> d.values()              # List of current values.
['bozo', 1234, 34]

>>> d.items()       # List of item tuples.
[('user','bozo'), ('p',1234), ('i',34)]
```

Logical Operators

- **You can also combine Boolean expressions.**
  - *True* if a is True and b is True: `a and b`
  - *True* if a is True or b is True: `a or b`
  - *True* if a is False: `not a`

if Statements (as expected)

```
if x == 3:
    print "X equals 3."
elif x == 2:
    print "X equals 2."
else:
    print "X equals something else."
print "This is outside the 'if'."
```

Note:

- Use of indentation for blocks
- Colon (:) after boolean expression

Selection – if, elif, and else

```
moons = 3
if moons < 0:
    print ‘less’
elif moons == 0:
    print ‘equal’
else:
    print ‘greater’
```

Always starts with if and a condition

There can be 0 or more elif clauses

The else clause has no condition and is executed if nothing else is done

Tests are always tried in order

Since moons is not less than 0 or equal to zero, neither of the first two blocks is executed

if *condition*:
    statements
[elif *condition*:
    statements] ...
else:
    statements

while *condition*:
    statements

for *var* in *sequence*:
    statements

break

continue

Repetition - Loops

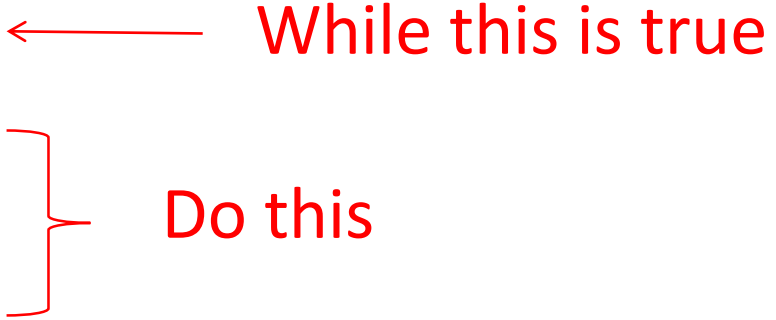
- Simplest form of repetition is the while loop

```
numMoons = 3
while numMoons > 0:
    print numMoons
    numMoons -= 1
```

Repetition - Loops

- Simplest form of repetition is the while loop

```
numMoons = 3
while numMoons > 0:
    print numMoons
    numMoons -= 1
```



Repetition - Loops

```
>>>numMoons = 3
>>>while numMoons > 0:
    print numMoons
    numMoons -= 1
3
2
1
>>>
```

For Loops 1

- For-each is Python's *only* form of for loop
- A for loop steps through each of the items in a collection type, or any other type of object which is "iterable"

```
for <item> in <collection>:
  <statements>
```

- If <collection> is a list or a tuple, then the loop steps through each element of the sequence.

- If <collection> is a string, then the loop steps through each character of the string.

```
for someChar in "Hello World":
    print someChar
```

For loops and the *range()* function

- We often want to write a loop where the variables ranges over some sequence of numbers. The *range()* function returns a list of numbers from 0 up to but not including the number we pass to it.

- range(5) returns [0,1,2,3,4]
- So we can say:

```
for x in range(5):
    print x
```

- (There are several other forms of *range()* that provide variants of this functionality…)
- *xrange()* returns an *iterator* that provides the same functionality more efficiently

Functions

```
def name(arg1, arg2, ...):
    """documentation"""          # optional doc string
    statements

return                           # from procedure
return expression                # from function
```

```
class Stack:
    "A well-known data structure…"
    def __init__(self):                 # constructor
        self.items = []
    def push(self, x):
        self.items.append(x) # the sky is the limit
    def pop(self):
        x = self.items[-1]              # what happens if it’s
empty?
        del self.items[-1]
        return x
    def empty(self):
        return len(self.items) == 0   # Boolean result
```