Introduction to Python 3

BY PETRI KUITTINEN 2020

What is Python?

- Interpreted, interactive programming language with dynamic typing
- Multi-paradigm:
  - structured
  - object-oriented
  - functional
- Free, open source
- Very popular
  - #2 in Github Octoverse 2019
  - #3 in Tiobe Index 8/2020
  - #1 Most wanted programming language Stack Overflow 2020
  - #1 Spectrum IEEE 2020
- Originally developed by Guido Van Rossum in 1991

Python Advantages

- Clean and elegant design
- Very easy to read, write and learn
- About 10 times faster development speed and shorter programs compared to traditional programming languages like C & Java
- Automatic memory management (reference count)
- Interactive interpreter
- Versatile inbuilt library and thousands of free modules from the net
- Good online documentation and excellent books
- Excellent glue language ("What Can I Do for You")

Python Disadvantages

- Much slower than C or Java for calculation-intensive applications
  - You can use libraries written in C/C++/Fortran to alleviate this
  - PyPy Just-in-Time (JIT) compiler written in subset of Python offers speedup
- Compulsory indentation of code is disliked by some people
- Because of dynamic nature, some errors aren't discovered before run time
- Not the optimal choice for mobile development

Python Usage

- A general purpose programming language
- Numerical, scientific and statistical computation, data analysis and visualization e.g. NumPy, SciPy, Pandas, Matplotlib, VisPy and Jupyter
- Artificial intelligence, machine learning e.g. TensorFlow and Scikit-Learn, Keras, Pytorch
- Web: Google search engine, YouTube, Instagram, Dropbox, Quora, Reddit, Yahoo, Spotify
- Games: Civilization IV, Sims 4, World of Tanks, Frets on Fire, Eve Online

Key Differences Between Python 2 and 3

PYTHON 2

```
print "hello"
print "begin",;print "end"

u"Unicode string"

3/2 # integer division
float(3)/2 # float div.

name = raw_input("?")
xrange(1,10)
```

PYTHON 3

```
print("hello")
print("begin",
end="");print("end")
"Unicode string"

3//2 # integer division
3/2 # float division

name = input("?")
range(1,10)
```

Python Keywords

- `False` `True` `and` `assert`
  `break` `class` `continue` `def`
  `del` `elif` `else` `exec`
  `finally` `for` `from` `global`
  `import` `in` `is` `lambda`
  `not` `nonlocal` `pass` `raise`
  `return` `try` `while` `with`
  `yield` `None`

- These cannot be used as names.
- In addition to these words Python has in-built functions, which shouldn't be used for names either.

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Python Names

- Use letters A-Z, a-z, numbers 0-9 and underscore (_) for names
- Capitalize class names: `ClassName, Matrix`
- Lowercase function and variable names: a, i, `load_settings`
- Uppercase constants: `MAXIMUM_WIDTH`
- Generally use underscore in variable and function names to separate words, but camel-case can also be sometimes used: `setWidth()`
- Use descriptive names
- Avoid names, which can cause confusion:
  `l (mistake with 1)`
  `O (mistake with 0)`

Python 3 Data Types

• int	unlimited precision integers	313
• float	double precision floating point	3.14
• complex	complex numbers	3+2j
• boolean	truth value	True or False
• str	string	"hello"
• list	unlimited array	[1, 2, "hello"]
• tuple	immutable version of list	(1, 2)
• dictionary	hash table	{"key": value}
• range	immutable sequence of numbers	range(0, 100, 10)
• set	like list, but no duplicates	{1, 2, 13}

- In addition there are other in-built types frozenset, bytes, bytearray, memoryview

Python Data Type Conversions

- Implicit type conversion only works for numeric types: integer → float → complex
- All other type conversions must be done manually:

```
s = "Answer is "+42 # error
s = "Answer is %d" % 42
s = "Answer is {}".format(42)
```

- `a = list(range(1, 6)) # [1, 2, 3, 4, 5]`
- `x = float(s) # will raise ValueError is s is # not a valid floating point number`

Operators and Their Evaluation Order

- , […] {…} '...'tuple, list & dictionary creation, string conversion
- s[i] s[i:j] s.attr f() indexing, slicing, attributes, function calls
- +x –x ~x positive sign, negative sign, binary not
- x**y raise x to power of y (x^y)
- x*y x/y x%y multiplication, division, modulo
- x+y x-y addition, subtraction
- x<<y x>>y shift integer x y bits left/right
- x&y binary and
- x^y binary exclusive or (XOR)
- x|y binary or
- Comparisons
- not x
- x and y
- x or y
- lambda args: expr anonymous function

Assignment Operators

- `a = b` assign object b to label a
- `a += b` roughly same as a = a + b
- `a -= b` roughly a = a - b
- `a *= b` roughly a = a * b
- `a /= b` roughly a = a / b
- `a //= b` roughly a = a // b
- `a %= b` roughly a = a % b
- `a **= b` roughly a = a ** b
- `a &= b` roughly a = a & b
- `a |= b` roughly a = a | b
- `a ^= b` roughly a = a ^ b
- `a <<= b` roughly a = a << b
- `a >>= b` roughly a = a >> b

Comparisons

- `<` lesser than
- `>` greater than
- `==` equal
- `!=` not equal (also `<>`)
- `<=` lesser than or equal
- `>=` greater than or equal
- `is [not]` object identity (same object)
- `[not] in` test if element is in sequence
- `a <= x <= b` x is between a and b

Boolean Operators

- `not x` (same as !x in C)
- `x or y` (same as x || y in C)
- `x and y` (same as x && y in C)
- Shortcut rules and automatic return:
  - If left side of OR operator is true, then the rest of boolean expression is not evaluated. Returns left if it is true, otherwise right side.
  - If left side of AND operator is false, then the rest of the boolean expression is not evaluated. Returns right side if left side is true, otherwise left side.
- None, numeric zero, empty sequences and mappings are considered to be False (anything else is true)
- For advanced coders, see also and-or trick

Integers

- Python 2 made a difference between:
  - `123 # normal integer`
  - `12345678901234567890L # long integer`
- Python3 automatically promotes integers to long integers e.g. L is not needed at the end of a very large integer.
- Hexadecimals (16-base): `0x41 = 65 0xff = 255`
- Binary (2-base): `0b111 = 7 0b1010 = 10`
- Octal (8-base): `0o10 = 8 # 010 in Python 2`

Floating Point Numbers

- IEEE 754 64-bit double precision
- $\pm 1.8 \times 10^{308}$
- About 15-17 of significant digits
- `3.141592653589793`
- `1e6 = 1000000.0`
- `-5e-3 = -0.005`
- `1.23e3 = 1230.0`
- Note that IEEE 754 double precision is inexact
- `0.2+0.7 = 0.8999999999999999`

Strings

- Both ' and " work and can be mixed e.g. `'easy '+"don't you think?"`
- The equivalent of Java's char is string with just one character `'a'`
- In Python 3 all strings are Unicode. In Python 2 strings are 8-bit and you must mark Unicode e.g. `u'this is Unicode'`
- Raw string make it easier to write regular expressions `two_numbers = r'\d\d'`
- Three quotes (''') or (""") make it easy to write multi-line string literals
- ```
  msg = """this is line #1
  line #2"""
  ```
- + concatenates two strings, but often it is not needed:
- `s = "eat " "my shorts" #same as s="eat "+"my shorts"`
- Strings can be indexed and indexing starts from 0
- String is Immutable Sequence Type, meaning its value cannot be changed after creating
- `a = "bart"; a[0]='c' # won't work`
- `a = "bart"; a = 'c'+a[1:] # will work, a = 'cart'`
  ```

String Literals

- "" or '' empty string
- *newline* backslash and newline ignored
- \\ backlash (\)
- \' single quote (')
- \" double quote (")
- \n ASCII linefeed (LF)
- \r ASCII carriage return (CR)
- \t ASCII horizontal tab (TAB)
- \xhh character with hex value hh
- \N{name}Character named name in the Unicode database (4)
- \uxxxx Character with 16-bit hex value xxxx (5)
- \Uxxxxxxxx Character with 32-bit hex value xxxxxxxx

String Methods

- **str.count(sub[, start[, end]])** Return the number of non-overlapping occurrences of substring sub in the range [start, end].
- **str.encode(encoding="utf-8", errors="strict")** Return an encoded version of the string as a bytes object. Default encoding is 'utf-8'. The opposite is **bytes.decode(encoding="utf-8", errors="strict")**
- **str.endswith(suffix[, start[, end]])** Return True if the string ends with the specified suffix, otherwise return False. suffix can also be a tuple of suffixes to look for.
- **s.find(substring)** find index of substring. Returns -1 if nothing found.
- **s.format(arguments)** formats a string for nicer output

String Methods I

- **str.isalnum()** Return True if all characters in the string are alphanumeric and there is at least one character, False otherwise.
- **str.isalpha()** Return True if all characters in the string are alphabetic and there is at least one character, False otherwise.
- **str.isdecimal()** Return True if all characters in the string are decimal characters and there is at least one character, False otherwise. Decimal characters are those that can be used to form numbers in base 10.
- **str.isdigit()** Return True if all characters in the string are digits and there is at least one character, False otherwise. Digits include decimal characters and digits that need special handling, such as the compatibility superscript digits.
- **str.islower()** Return True if all cased characters in the string are lowercase and there is at least one cased character, False otherwise.
- **str.isnumeric()** Return True if all characters in the string are numeric characters, and there is at least one character, False otherwise.
- **str.isspace()** Return True if there are only whitespace characters in the string and there is at least one character, False otherwise.
- **str.isupper()** Return True if all cased characters in the string are uppercase and there is at least one cased character, False otherwise.

String Methods J-U

- **`s.join(sequence)`** concatenate sequence with string in between
- **`s.lower()`** convert to lowercase
- **`s.replace(old, new)`** replace old with new
- **`s.split(separator)`** split into a list
- **str.splitlines([keepends])** Return a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true.
- **str.startswith(prefix[, start[, end]])** Return True if string starts with the prefix, otherwise return False. prefix can also be a tuple of prefixes to look for.
- **`s.strip()`** remove white space
- **`s.upper()`** convert to uppercase

Different Ways of String Formatting

- The old syntax (works with any Python version):
  `"string" % parameter  "string" % (parameter1, parameter2)`
- `"Value of %s is %.2f" % ("PI", 3.14159)`
- The new syntax (Python 2.6+):
- `"Value of {0} is {1:.2f}".format("PI", 3.14159)`
- Python 3.6+ f strings:
- `name = "pi"; value = 3.14159`
  `f"Value of {name.upper()} is {value:.3f}"`

String Formatting

```
# printing numbers with certain amount of decimals
x = 123.45678585
print("2 decimals %.2f and 3 decimals %.3f ." % (x, x))
# using named arguments
age = 15
name = "Jack"
print("My name is %(name)s and my age is %(age)d." % {"age":
age, "name": name})
# or if you are lazy
print("My name is %(name)s and my age is %(age)d." % vars())
# formatting to binary and hexadecimal
print("%o %x" % (13, 65))
```

String Examples

- `"The End".lower()  # 'the end'`
- `"The End".upper()  # 'THE END'`
- `s = "test"; s.capitalize()  # doesn't change s`
  `s = "test"; s = s.capitalize() # changes s`
- `b = s.encode("utf-8")  # encode string into bytes object`
  `s = b.decode("utf-8")  # decode bytes object into string`
- `words = "This is easy".split()  # ['This', 'is', 'easy']`
  `sentence = " ".join(words)  # Join the strings into one`
- `" get rid of leading and trailing white space \n".strip()`
- `"fun" in "This is funny"`
- ```
  s = "line1\nline 2 is longer\nline 3 is even longer\n"

  for line in s.splitlines():
      print(line)
  ```
  ```

Practical String Examples

- ```
  name_input = "  John Smith "
  name = name_input.strip()
  ```
- ```
  s = 'programming'
  r = s[::-1]  # s reversed 'gnimmargorp'
  ```
- ```
  phone_input = " +358 45 123 8976 "
  phone = phone_input.replace(" ", "")   # '+358451238976'
  ```
- ```
  file_paths = ['pic.jpg', 'text.text', 'logo.png']
  file_endings = ('.png', '.gif', '.jpg', '.jpeg')
  for path in file_paths:
      if path.endswith(file_endings):
          print(path)  # print all image file paths
  ```

Lists

- List is a mutable sequence type
- List can contain any other python objects, including other lists
- List can change its contents and size
- Implemented as resizing array of pointers
  - fast random access and change of elements
  - slow adding or removing items to beginning of a large list (use deque from collections module for that)
- `a = [0, 2.5, "cat"]`
- `a[0] = 1`
- `b = [] # empty list`
- `c = [0]*100 # list with 100 zeros`

Tuples

- Immutable aka read-only list, thus hashable and can be used as key for dictionaries
- ```
  () # Empty tuple
  (1, ) # one element tuple
  (1, 2, "cat") # three element tuple
  ```
- f = (1, 1, 2, 3, 5, 8)
  f.count(1) # 2, counts how many 1:s it can find from f
  max(f) # 8 largest element
  sum(f) # 20 sum of all elements
  ```

Mutable Sequence Operations

- `s[i] = x` item i of s is replaced by x
- `s[i:j] = t` slice of s from i to j is replaced by the contents of the iterable t
- `del s[i:j]` same as s[i:j] = []
- `s[i:j:k] = t` the elements of s[i:j:k] are replaced by those of t
- `s *= n` updates s with its contents repeated n times
- `s.append(x)` appends x to the end of the sequence (same as s[len(s):len(s)] = [x])
- `s.insert(i, x)` inserts x into s at the index given by i (same as s[i:i] = [x])
- `s.pop([i])` retrieves the item at i and also removes it from s
- `s.remove(x)` remove the first item from s where s[i] is equal to x
- `s.reverse()` reverses the items of s in place
- `sort(*, key=None, reverse=False)` sorts the list in place

Sequence Operations

- `x in s` True if x is in s
- `x not in s` False if x is in s
- `a+b` concatenation
- `s*n` (shallow) copy n times s
- `s.count(x)` return total number of occurrences of x in s
- `s[i]` return i:th element of s, indexes start from 0
- `s[i:j]` slicing from i to j (not including)
- `s[i:j:k]` slice from i to j with stepping k
- `len(s)` length/size of s
- `max(s)` largest item of s
- `min(s)` smallest item of s

Examples of Slicing

- [begin:end:stepping], indexes start from 0, negative indexes from -1 start from the end
- These will work with any sequence type e.g. string, list, tuple
- ```
  s = "abc123"
  print(s[2:])     # omit first 2 characters "c123"
  print(s[-1])     # last character "3"
  print(s[:-3])    # omit last 3, "abc"
  print(s[2:5])    # "c12"
  print(s[::2])    # every second character "ac2"
  print(s[::-1])   # reverse "321cba"
  a = ["one", "two", "three", "four"]
  print(a[-2])     # second last, "three"
  print(a[::-1])   # ['four', 'three', 'two', 'one']
  a[1:3] = ["kaksi", "kolme"]
  print(a)         # ['one', 'kaksi', 'kolme', 'four']
  ```
  ```

Examples of Sequence Operations

```
s = "this is easy"
n = len(s) # n = 12
print(s.count("i"))   # how many "i":s
words = s.split()
sentence = " ".join(words)
print(words, sentence)
if words == sentence:
    print("equal")
a = []
a.append(1)
a.append(2)
a.append(100)
print(a[1])           # print second item
print(a.pop())        # print last item and remove it
print(a.pop(0))       # print first item and remove it
a.insert(1,"second")  # insert to specified location
```

Examples of List Operations

```
from random import shuffle
a = list(range(1,11))  # a = [1, 2, 3…10]
print("Sum:", sum(a))  # 55
print("Smallest:", min(a))
print("Largest:", max(a))
shuffle(a)
print("Shuffled;", a)
a.sort()
print("Sorted:", a)
a.reverse()
print("Resersed", a)
# more efficient way to do reverse sort
a.sort(reverse=True)
```

Copy by Reference vs Shallow Copy

- ```
  a = [1, 2, -4, 100]
  b = a # b is now reference to a, not a copy!
  b[0] = "flower"
  print(b)
  print(a) # both are changed, because they point to sam
  e memory location
  ```
- ```
  # solution to this problem
  print("Do like this instead:")
  a = [1, 2, -4, 100]
  b = a[:] # b is now a shallow copy of a
  # b a.copy() would do the same
  b[0] = "flower"
  print(b) # 'flower' here
  print(a) # no 'flower' here
  ```

Set and frozenset

- A set object is an unordered collection of distinct hashable objects.
- Set has no duplicate items, unlike list
- frozenset is immutable version of set
- `s = {item1, item2, item3}`
- `s = set([iterable])`
- **issubset(other)** Return true if set is subset of other
- **union(*others)** Return a new set with elements from the set and all others.
- **intersection(*others)** Return a new set with elements common to the set and all others.
- **difference(*others)** Return a new set with elements in the set that are not in the others.
- **symmetric_difference(other)** Return a new set with elements in either the set or other but not bot

Using Sets

```
a = {1, 2, 3, 4, 5}
b = {10, 8, 7, 5, 4}
c = a.union(b)
d = set([7, 5])
print("length", len(a))
if 3 in a:
      print("3 found in a")
print("union", c)
if a.issubset(c):
      print("a is subset of c")
print("intersection", a.intersection(b))
print("difference", a.difference(b))
print("symmetric difference", a.symmetric_difference(b))
readonly = frozenset([1, 2, 3])
```

Dictionaries

- Is a built-in mapping type, which maps immutable values (numbers, strings, tuple) to arbitrary objects
- Dictionaries are mutable
- Syntax: `{key: value}`
- Implemented as a hash table
  - In most cases very fast random access
- `d = {} # empty dictionary`
- `d = {"Jack": 1234, "Lisa": 6678 }`
- `d["Lisa"] # return 6678`
- `d["Lisa"] = 6699`
- `d["Bob"] = 101`

Dictionary operations

- `len(d)` size of dictionary
- `d[k]` value of a with key k
- `d[k] = v` set a[k] to v
- `del d[k]` remove a[k]
- `d.clear()` empty dictionary
- `k in d` True if a has k
- `d.has_key(k)` True if a has k
- `d.get(key [, default])` return the value for key if key is in dictionary, else return default
- `d.items()` a copy of a's (key, value) tuples as a list
- `d.keys()` a copy of a's keys as a list, same as list(d)
- `d.setdefault(key [,default])` if key is in dictionary, return its value. If not insert key with value default and return default
- `d.values()` a copy of a's values as a list

Dictionary Examples

```
d  = {} # empty dictionary
d = {"cat": "kissa", "dog": "koira"} # define a dictionary with 2
 key value pairs
print(d)
d["ape"] = "apina" # add a new key value pair
print(d["cat"])
d["cat"] = "mirri" # update existing key value pair
print(d["cat"])
if "cat" in d: # checking if something exists in a dictionary
    print("cat found")
for key in d.keys():
    print(key)
for value in d.values():
    print(value)
for word, definition in d.items():
    print(word, "=", definition)
for word in sorted(d):
    print(word)
del d["dog"] # remove
```

Dir, Sorted and Vars

- `dir([object])` function returns a list of valid attributes of the object.
  dir() alone returns a list of objects in current name space
- `sorted(iterable, key=None, reverse=False)`
  function sorts the elements of a given iterable in a ascending order (descending order if reverse is True) and returns the sorted iterable as a list
- `vars(object)`
  function returns the __dict__ attribute of the given object.
  Basically vars() returns current namespace as dictionary

Python Interactive Interpreter Command Line

- Cursor up and down to browse command line history
- Tab for autocompletion
- `import random # first import random module, as an example`
- `dir(random) # print everything from random name space`
- `dir() # global name space`
- `help() # interactive help, inside help type topics`
- `help('topic') # help on specific topics or quit to end`
- `help('string') # information about strings`
- `help('random') # information about random module`
- `print(random.__doc__) # will do almost the same`
- `quit() or Control-D will quit the interpreter`

Basic Input and Output

- `input(string)  # returns a string`
- `print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)`
- `print("automatic newline")`
- `print("no newline", end="")`
- `print("automatic space", "between")`
- `print(2, 3, sep=" and ") # '2 and 3'`

Interactive Hello World

```
# Python 3 version
name = input("What is your name?")
print("Hello", name)

#!/usr/bin/python
# -*- coding: utf-8 -*-
# Python 2 generic version
name = raw_input("Mikä sinun nimesi on?")
print "Hei "+name
```

Comments and Documentation Strings

- # single line comment to the end of line
- modules (=files), classes and functions can have a documentation string, which is a just string at the beginning. It can be accessed name.__doc__
- ```
  def square(x):
      """return x raised to power of two"""
      return x*x
  ```
- You can mark encoding with a special comment (needed in Python2, which uses ASCII as default encoding!)

```
# -*- coding: utf-8 -*-
```
```

Block Are Expressed as Indentation

- C/C++/C#/Java/JavaScript use braces to designate start and end of block. Indentation is optional.
- In Python indentation is compulsory. You should use standard indentation of 4 spaces, but any amount will work.
- Use PEP8 as style guide
- ```
  for i in range(1, 11):
      print(i)  # inside the loop
  print("end")  # outside the loop
  ```
  ```

If statement

Syntax:

```
if condition:
    # executed if condition is true, end of if-else
elif condition:
    # else if, executed if above not true and this is true
else:
    # executed if all above is false

# Notice:
if 1 <= age <= 10:
    print("Your age is between 1-10")
```

If example

```
age = float(input("Age?"))
if age <= 0:
    print("Nobody can be so young.")
elif age < 10:
    print("You are still young.")
else:
    print("You old fart!")
# common mistakes:
# 1. forgetting colon : at the end
# 2. wrong indentation
# 3. trying to use = for comparison, use == instead
# e.g. if a=3: fails, but if a == 3: works
```

Ternary Operator aka inline if-else

```
x = float(input("Give me number?"))
# ternary operator is nice for short ifs
abs_x = x if x >= 0 else -x
# same as:
if x <= 0:
    abs_x = x
else:
    abs_x = -x

# can also be used in functions and lambdas
def myabs(x):
    return x if x >= 0 else -x

print("Absolute value is", myabs(x))
```

Writing Long and Complex If Clauses

```
if isinstance(s, str) and len(s)>0 and not " " in s:
    # verifying that s is a strong of not empty
    # value and that it doesn't contain any space

# consider making a function

def valid_string(s):
    return isinstance(s, str) and \
    len(s) > 0 and not " " in s
if valid_string(s):

# the above is more readable and reusable
```

While loop statement

- basic loop construct, same as C/Java/JavaScript except for optional else part, which is executed when loop ends (unless user breaks out of it with break)
- Syntax:

```
while condition:
      statement(s)
[else: statement(s)]
```

```
# Common mistake is to forget to change value of number
```

```
# print numbers from 1...10
x = 1
while x <= 10:
    print(x)
    x += 1  # same as x = x+1
print("end")
```

There is No Do While

- Use the following construct instead:

```
while True:
    statement(s)
    if not condition: break
```

- Same as C/Java/JavaScript's:

```
do {
    statement(s);
} while (condition);
```

Do While Example

```
x = 1
# emulate do while with endless while and break
while True:
    print(x)
    x += 1
    if x>10: break
```

For Loop Statement

- Iterates over a sequence, which can be a list, tuple, range, string, file (returns one line)
- Optional else part is executed when loop ends or normally or if there was no elements in sequence

Syntax:

```
for element in sequence:
    statement(s)
[else: statements ]
```

```
for i in range(10):
    print(i)
```

```
for x in [1, "cat", 3.14]:
    print(x)
```

For - Else

```
words = input("Give me words separated by spaces, but n
ot word NO?").split()
for word in words:
    if word.lower() == "no": break
    print(word)
else:
    # reached if no words or no break
    print("That was nice!")
```

Reverse loop

```
names = ["Bill", "James", "Paul", "Paula", "Jenny", "Kate"]

print("Normal order:")
for name in names:
    print(name)

print("Reverse order:")
for name in reversed(names):
    print(name)

print("This will also do the same, but less efficient:")
for name in names[::-1]:
    print(name)

print("Don't code like this:")
i = len(names)-1
while i >= 0:
    print(names[i])
    i = i-1
# this also works, but unreadable
for i in range(len(names)-1, -1, -1):
    print(names[i])
```

Nested Loops

```
N = 10
for y in range(1, N+1):
    for x in range(1, N+1):
        print("%3d " % (x*y), end="")
    print()

#output:
  1   2   3   4   5   6   7   8   9  10
  2   4   6   8  10  12  14  16  18  20
  3   6   9  12  15  18  21  24  27  30
  4   8  12  16  20  24  28  32  36  40
  5  10  15  20  25  30  35  40  45  50
  6  12  18  24  30  36  42  48  54  60
  7  14  21  28  35  42  49  56  63  70
  8  16  24  32  40  48  56  64  72  80
  9  18  27  36  45  54  63  72  81  90
 10  20  30  40  50  60  70  80  90 100
```

Break and continue

- `break` exits from for or while loop
- `continue` continues the nearest cycle of for or while loop (goes to beginning of loop)
-
```
while True:
    s = input("positive integer?")
    if not s.isdigit():
        continue
    i = int(s)
    if i>0:
        break
```

Def - Function definitions

- Syntax:

```
def function_name(parameter_list):
    ["documentation string"]
    # function_body
    return value(s)
```

- *return* keyword ends function/method. It can have one or many values (tuple) or no values at all (when it returns *None)*

- ```
  def is_palindrome(s):
      """returns True if s is palindrome"""
      s = s.lower()
      return s == s[::-1]
  ```
  ```

Calling a Function

```
value = function_name(function_args)
```

- Examples

```
s = input("Give me a word: ")
if is_palindrome(s):
    print(s, "is palindrome")
else:
    print(s, "is not a palindrome")
# functions can be also referenced via variable
new_name = is_palindrome
print(new_name(input("word?")))
```

Different Kind of Functions

```
# function which takes arguments and returns something
def inches_to_cm(inch):
    return inch * 2.54

# function which takes arguments, but doesn't return anything
def print_many(s, n):
    for i in range(n):
        print(s)

# function which has no arguments, but has a return value
def tell_secret():
    return "The secret is you."

# function without parameters nor return value
def the_end():
    print("This is the end.")

print(inches_to_cm(10))
print_many("hello", 5)
print(tell_secret())
the_end()
```

Default Arguments

```
def ask(question, choices, correct, retries=2):
    print(question)
    for i, choice in enumerate(choices, start=1):
        print(i, choice)
    while retries > 0:
        try:
            guess = int(input("?"))
        except ValueError:
            continue
        if guess == correct:
            print("Correct!")
            break
        print("Wrong guess")
        retries -= 1
    else:
        print("The correct reply was choice", correct, "=", choices[correct - 1])

ask("What is the capital of Australia?", \
    ("London", "Sydney", "Canberra", "Victoria"), 3)
ask("When Finland gained independence?", \
    ("1900", "1917", "1919", "1939"), 2, 1)
```

Keyword Arguments

- When calling a function you can you can specify the name of the arguments with *name=value*. These are called *keyword arguments.* The keyword arguments can then appear in different order than using positional function arguments. This is especially useful if you have trouble remembering the order of arguments.

```
ask(question = "What is the chemical symbol of Iron?", \
    correct=1, choices=("Fe", "R", "Ir", "I"))
```

- There can't be any positional arguments after keyword arguments, so the following does NOT work:

```
ask(question="How to delete a variable in Python?", \
    retries=3, choices=("delete", "del", "remove", "destroy"), 2)
```

- But the following will work:

```
ask("How to delete a variable in Python?", \
    ("delete", "del", "remove", "destroy"), \
    retries=3, correct=2)
```

Variable Amount of Function Arguments

- * before argument will return a tuple of 0..N arguments

```
def mysum(*n):
    total = 0
    for i in n:
        total += i
    return total
print(mysum(1, 2, 10))
# sum([1, 2, 10]) would do the same
```

Variable Amount of Keyword Arguments

- ** before argument will return a dictionary of 0..N keyword arguments

```
def login(**parameters):
    if "user" in parameters:
        print("User name:", parameters.get("user"))
    if "password" in parameters:
        print("Password:", parameters.get("password"))

login(user="root", password="qwerty")
# Note that the following will fail, because it has
# no keyword arguments
# login("root", "qwerty")
```

Lambda - Small Anonymous Function with 1 expression

- Syntax:

```
lambda arguments: expression
```

- Almost same as creating:

```
def anonymous(arguments):
    return expression
```

-
```
f = lambda x: x+1
print(f(5))  # 6
```

-
```
product = lambda a, b: a*b
print(product(3,2))  # 6
```

-
```
my_abs = lambda x: x if x >= 0 else -x
print(my_abs(-5))  # 5
```

Functions Returning Functions

- Since Python functions are first-class objects, they can be used like variables or as return values.

```
def make_adder(n):
    def adder(x):
        return x+n
    return adder

# More concise version using lambda
def make_adder(n):
    return lambda x: x+n

inc2 = make_adder(2)
print(inc2(4))
```

List Comprehensions

- List comprehensions provide a concise way to create lists.
- It consists of brackets containing an expression followed by a for clause, then 0… N for or if clauses.

```
a = [1, 2, 3, 10, -5, 7]
squared = [x*x for x in a]
odd = [x for x in a if x%2 != 0]
b = ['1', "2", '3', "100", "-15",
"1234", "1000000", "987654"]
int_list = [int(i) for i in b]
big_ints = [int(i) for i in b if len(i) > 3]
```

Functional Programming - map, reduce, filter

- **map(function, sequence)** applies function to all items of a sequence
- **filter(function, sequence)** creates a new sequence consisting of those items, which the function returned true
- **reduce(function, sequence)** reduce the sequence to one result by applying given function to list

```
# in Python 2 reduce in built-in function
# in Python 3 you must import reduce from functools
from functools import reduce
a = list(range(1, 11))
b = list(map(lambda x: x * x, a))  # square items
c = list(filter(lambda x: x % 2 == 0, a))  # select even
d = reduce(lambda x, y: x + y, a)  # calculate sum
# chain b, c, d
e = reduce(lambda x, y: x + y,
           map(lambda x: x * x, filter(lambda x: x % 2 == 0, a)))
# same with list comprehension
sum([x * x for x in a if x % 2 == 0])
```

Recursion

- A function calling itself. Useful for some algorithms.

```
# Example of loop using tail recursion
def f(n):
    print(n)
    if n > 0: f(n-1)
f(10)  # print from 10 to 0

def factorial(n):
    if n == 0: return 1
    if n > 0: return n * factorial(n-1)

print(factorial(6))  # 6*5*4*3*2*1 = 720
```

Exceptions

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions.*
- Common Python exceptions:
  - `IndexError` index out of range
  - `NameError` name not defined
  - `OSError` operating system related error e.g. error accessing a file
  - `SyntaxError` erroneous syntax
  - `TypeError` cannot concanate different types
  - `ValueError` invalid type
  - `ZeroDivisionError` division by zero

This Is What Happens without Exception Handling

```
>>> int("not a number")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base
10: 'not a number'
>>> 2/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> a = [1, 2, 3]
>>> a[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Handling Exceptions

```
try:
    # if an exception is fired in this block go to
    # the corresponding exception handler
except ExceptionClass:
    # handle this exception type
except AnotherExceptionClass:
    # handle this exception type
except:
    # catch all exceptions, NOT recommended!
else:
    # executed if no exceptions occured
finally:
    # executed at the end, no matter what happens

raise ExceptionClass(message) # trigger exception manually
raise # raise exception again
```

Try Except Example

```
import sys
try:
    with open('myfile.txt') as f:
        s = f.readline()
        i = int(s.strip())
        print(i)
except OSError as err:
    print("OS error:", err)
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

Exception Else Example

```
import sys
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

Safe Way to Ask Floating Point Numbers from User

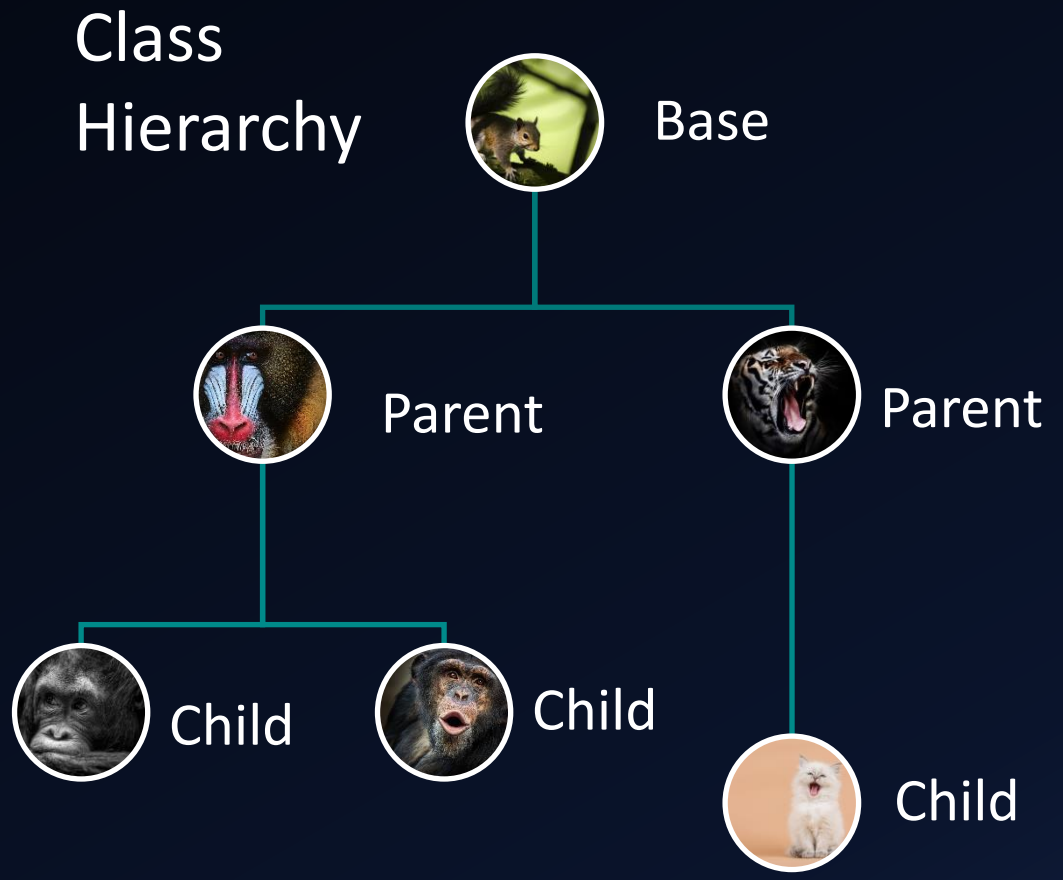
```
def ask_float(question):
    """display the question string and wait for standard input
    keep asking the question until user provides a valid floating
    point number. Return the number"""
    while True:
        try:
            return float(input(question))
        except ValueError:
            print("Please give a valid floating point number")

def lbs_to_kg(lbs):
    """Convert pounds (lbs) to kilograms (kg)"""
    return lbs * 0.45359237

lbs = ask_float("How many pounds (lbs) ?")
print(f"{lbs_to_kg(lbs):.2f} kg")
```

Object Oriented Programming (OOP)

- **Child class / Child Class** = inherited from a parent class
- **Class** = definition of data and methods, a blueprint for class of objects
- **Encapsulation** = hiding data
- **Inheritance** = copying features from parent and its parents
- **Member or attribute of object** = feature of an object
- **Method** = function attached to an instance of a class
- **Method Overloading** = Method with same name, but with different number or type of parameters
- **Object** = Instance of a Class, a collection of data
- **Operator Overloading** = Change meaning of an operator



Object Oriented Programming in Python

- Python uses class-based object-oriented programming, which supports multiple inheritance and operator overloading.
- Python functions are first-class objects, meaning they can be stored in data structures.
- `class SubClassName(ParentClass1, ParentClass2…):`
- Everything is by default public:
  _ in front of name = protected
  __ in front of name = private
- Special methods use __specialname__:
  - `__init__(self, parameters)` constructor
  - `__str__(self)` return string representation of the object
- Creation of new objects doesn't use any new keyword:

```
new_object = ClassName(parameters)
# changing and reading public attributes is trivial
new_object.attribute = new_value
print(new_object.attribute)
```

Class Definition and Inheritance

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return f"Person('{self.name}', {self.age})"

class Student(Person):
    def __init__(self, name, age, id_no):
        # in Python2 super(Student, self).__init__(name,age)
        super().__init__(name, age)
        self.id_no = id_no
    def __str__(self):
        return f"Student('{self.name}', {self.age}, '{self.id_no}')"

jack = Student("Jack Student", 25, "10100")
jack.age += 1
print(jack)  # Student('Jack Student', 26, '10100')
```

Empty Classes Can be Useful

```
class Person:
    pass    # empty class acts as a dictionary like container

jack = Person() # create new instance of class Person, notice now new keyword
jack.name = "Jack Smith"
jack.age = 24
bob = Person()
bob.name = "Bob Paxton"
bob.age = 66
jill = Person()
jill.name = "Jill Taylor"
jill.age = 21
jill.address = "Example Street 12"
persons = [jack, bob, jill]
for person in persons:
    person.age += 1
    print(person.name, person.age, end='')
    if hasattr(person, "address"):
        print("", person.address)
    else:
        print()
```

Using Slots to Speed Up Classes

- By default Python uses a dictionary to store an object's instance attributes. This allows the ultimate flexibility by setting arbitrary new attributes at runtime, but it can also consume quite a lot of RAM and be quite slow.
- If small memory use and fast execution is your main consideration and you know all your instance attributes in advance, consider defining them at class level using:

```
__slots__ = ['attribute1', 'attribute2']
```

Slots Example

```
class MyClass(object):
    def __init__(self, name, identifier):
        self.name = name
        self.identifier = identifier

m = MyClass("name", "id")
m.age = 100  # You can add new identifiers
print(m.age)

# Same with slots
class MyClass2(object):
    __slots__ = ['name', 'identifier']
    def __init__(self, name, identifier):
        self.name = name
        self.identifier = identifier

m = MyClass2("name", "id")
m.age = 100  # will fail!
```

Operator Overloading

```
class Complex:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __add__(self, other):
        return Complex(self.real + other.real, self.imag + other.imag)

    def __str__(self):
        return f"{self.real}+{self.imag}i"

    def __repr__(self):
        return f"Complex({self.real}, {self.imag})"


x = Complex(1, 2)
y = Complex(2, 4)
x = x + y

print(x)  # calling __str__ for human readable version

print(repr(x))  # calling __repr__ for string which allows to recreate object
```

Type Hints = Static Type Checking for Python

- Python 3.6 introduced type hints. They don't affect run-time at all, but can improve code documentation and readability.
- Mypy, a popular Python type checker, will check the type hints:

```
pip3 install mypy
myxy test.py
```

-
```
variable: type  # weight: int
variable: type = value  # x: float = 13.5
collection: CollectionName[type…]  # List[str]
def func(arg1: type, arg2: type = val) -> returnval:
# def square(x: float) -> float:
```

- Built-in types: int, float, bool, str
- The following must be imported from typing module:
  - Collections: List, Tuple, Set, Dict, Sequence
  - Special: Any, Optional, Generic, Union, Callable, Iterator, Mapping, TypeVar

Using Type Hints

```
from typing import List, Dict, Tuple, Union, Any

def is_negative(x: float) -> bool:
    return x < 0

def print_many(s: Union[str, int, float], n: int = 5) -> None:
    for i in range(n):
        print(s)

def multiply_list(a: List[float], n: float) -> List[float]:
    return [n*i for i in a]

d: Dict[str, Any] = {"key": "value", "key2": ("tuple", ), "key3": 13}
#d[123] = "bug"
print(is_negative(4))
#print(is_negative("bug"))
print_many(5)
#print_many("bug", 3.5)
print(multiply_list([1, 2.5, 3], 2.5))
#print(multiply_list([1, "bug", 3.14], 5))
#print(multiply_list([1, "cat", 3.14], "bug"))
```

Data Classes and Type Checking Annotations

```
from dataclasses import dataclass

@dataclass
class Person:
    name: str
    age: int

@dataclass
class Student(Person):
    id_no: str

jack = Student("Jack Student", 25, "10100")
jack.age += 1
print(jack)
# Student(name='Jack Student', age=26, id_no='10100')
```

Operator Overloading with Data Classes and Forward Referencing

```
from __future__ import annotations  # for forward references
from dataclasses import dataclass

@dataclass
class Complex:
    real: float
    imag: float

    def __add__(self, other: Complex) -> Complex:
        return Complex(self.real + other.real, self.imag + other.imag)

x = Complex(1, 2)
y = Complex(2, 4)
x = x + y
print(x)
```

Iterators

- iterator is an object that can be iterated upon, meaning that you can traverse through all the values
- an iterator is an object which implements the iterator protocol, which consist of the methods **__iter__()** and **__next__()**.
- Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable containers which you can get an iterator from.
- All these objects have a **iter()** method which is used to get an iterator
- ```
  foods = ("carrot", "apple", "pear")
  my_interator = iter(foods)
  print(next(my_iterator))
  print(next(my_iterator))
  ```
  ```

Creating Your Own Iterator

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

a = [1, 2, 100, 30]
for i in Reverse(a):
    print(i)
```

Generators

- Generators provide an easier way to do iterators using the **yield** keyword instead of return in function definitions
- Generator functions allow you to declare a function that behaves like an iterator, i.e. it can be used in a for loop
- You can use generators to build objects and then iterate them using next(generator_object) just like you were using iterator objects
- Generators can also be used like list comprehensions:

```
# list comprehension
doubles = [2 * n for n in range(50)]
# same as the list comprehension above
doubles = list(2 * n for n in range(50))
```

Generator Examples

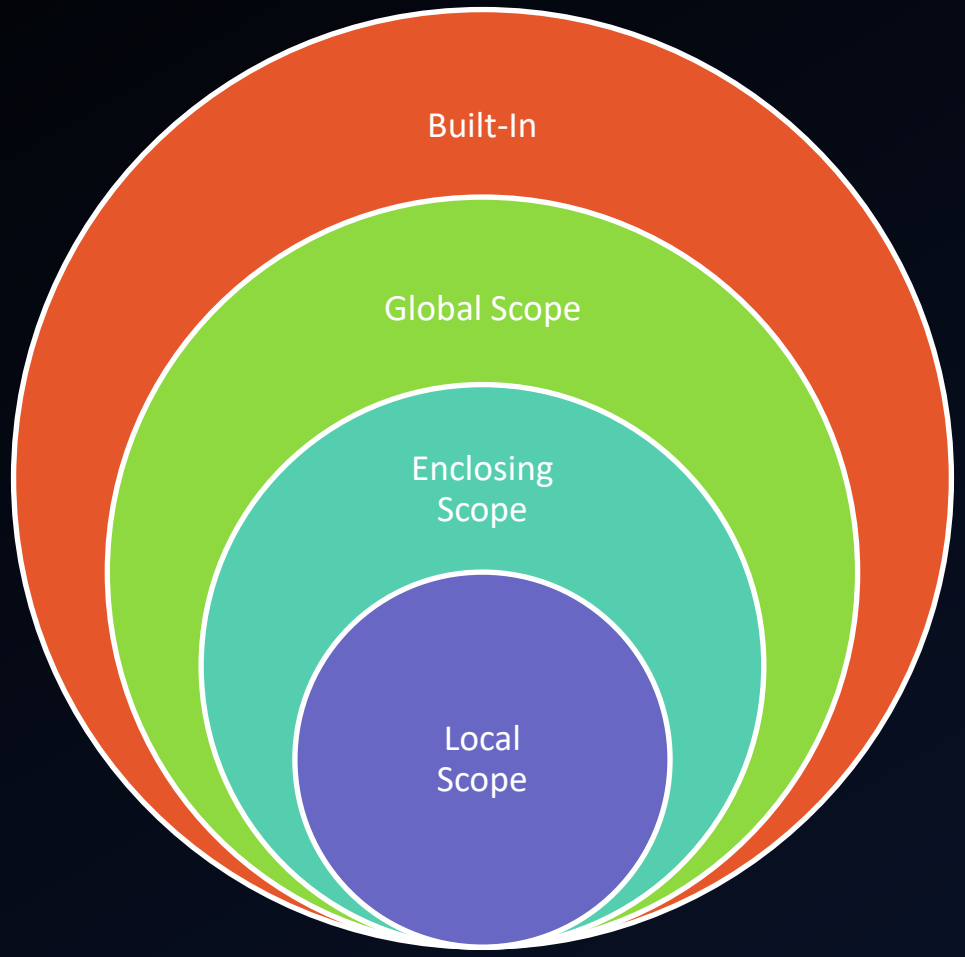
```
def reverse(data):
    """Generator iterating backwards"""
    for index in range(len(data)-1, -1, -1):
        yield data[index]

for char in reverse('spam'):
    print(char)

def fib():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

f = fib()
for x in range(2000):
    print(next(f))
```

Python Scopes



- Any name declared inside a function is local, unless preceded with keyword **nonlocal** (enclosing scope) or **global**
- **Local** is usually better than **Global**.
  - Faster
  - Less pollution of global name-space

Local vs Global

```
x = 2  # global x

def f(x):  # this parameter x is local
    x = x * 2  # this is local x
    y = x * 2  # this is local y
    return y

print("x =", x)
y = f(3)  # this is global y
print("y =", y)
print("x =", x)
x = 10  # this is global x
print("x =", x)
```

Local vs Nonlocal vs Global

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

Modules

- Every Python file (.py) is automatically a module, with its own name space.
- Before writing your own code, study / search the standard module.
- Name your Python file / module, using only the following characters: a-z, 0-9 and _ (do NOT use space, ÄÖ, capital letters etc.)
- Do NOT create Python files, which have the same name as:
  - keyword in python e.g. while, for, if, def, class, global
  - built-in function in python e.g. list, min, max, print, input, filter, map
  - a module in standard library e.g. os, sys, math, cgi, socket, re, random
- If you by mistake create a Python file with same name as inbuilt function or module in standard library, you are risking to mask / override that name.

Python Main Program

- If you import a Python file you created, it will run it as well.

If you are planning to make a library and not a main program file, then consider using:

```
if __name__ == "__main__":
    main()
```

- The above code is run only if the current file is invoked as main the program. In other words if above code was inside mycode.py, main() will be executed if:
  `python mycode.py`
  but main will not be executed if you just "import mycode" from another file

Example of a module - myutil.py

```
"""Miscellaneous utility functions"""

def ask_float(question):
    """Display the question string and wait for standard input.
    Keep asking the question until user provides a valid floating point number.
    Return the number."""
    while True:
        try:
            return float(input(question))
        except ValueError:
            print("You didn't give me a number")

def lbs_to_kg(lbs):
    """Convert pounds (lbs) to kilograms (kg)"""
    return lbs * 0.45359237

def main():
    """Main program to test the module"""
    print(lbs_to_kg(ask_float("Give me pounds?")))

if __name__ == "__main__":
    main()
```

Using myutil.py from Another Python Module / File

```
import myutil
# NOTICE: no .py needed here, just the part before .py
# if you import like this each module has its own name space
# and must be prefixed like below
x = myutil.ask_float("Give me a float?")
print(x)

from myutil import ask_float, lbs_to_kg
# if you import like this then the name is copied to the local name space
# and no prefix is required
x = ask_float("Give me a float?")
print(lbs_to_kg(x))

from myutil import *
# Copy all names from the module to current name space
# WARNING: Do NOT use this often, because local name space becomes polluted!
x = ask_float("Give me a float?")
print(x)

from myutil import ask_float as ask_number
# import with a new name
x = ask_number("Give me a float?")
print(x)
```

Python Packages

- A directory with Python code and __init__.py is a package
- Packages can contain subfolders and subpackages.
- Folders without __init__.py are ignored, but __init__.py can be empty
- ```
  import package.module
  package.module.func()
  from package import module
  module.func()
  from package.module import func
  func()
  ```
  ```

Using Packages

```
# mypackage/mymath.py
# mypackage/__init.__ also present

def average(a):
    """return arithmetic mean of given list a of numbers"""
    return sum(a)/len(a)


# from root folder using_package_import.py
import mypackage.mymath

numbers = (1, 3.14, 2)
print("Average:", mypackage.mymath.average(numbers))

from mypackage import mymath
print("Average:", mymath.average(numbers))

from mypackage.mymath import average
print("Average:", average(numbers))
```

Closure

- What is a closure? (you need to satisfy all three below)
  1. Closure is a nested function (inner function inside an outer function).
  2. The nested inner function must refer to a value defined in the enclosing function.
  3. The enclosing function must return the nested inner function.
- A closure is an instance of a function, a value, whose non-local variables have been bound either to values or to storage locations
- A closure is a value like any other value. It does not need to be assigned to a variable and can instead be used directly or so called "anonymous closure".
- Closures can avoid the use of global values and provides some form of data hiding. It can also provide an elegant solution to the problem, instead of making a class. If you need a lot of attributes and functionality class is usually better solution.

Closure Example

```
# Example of a closure
def make_multiplier(x):  # outer / enclosing function
    def multiplier(y):  # inner / nested function
        return x*y
    return multiplier

mul10 = make_multiplier(10)  # mul10 is the closure
print(mul10(5))  # 50
print(mul10(10))  #100
mul2 = make_multiplier(2)  # mul2 is the closure
print(mul2(10))  # 20
```

Decorators

- Decorators to add functionality to an existing code, using closures.
- This is also called metaprogramming because a part of the program tries to modify another part of the program at compile time.
- Defining and using decorators

```
def decorator_name(func):
    def inner(*args, **kwargs):
        # decorator does its job here
    return inner

# decorator will modify a function
@decorator_name
def something_else():
    # function body
    pass
```

Decorator Example

```
def safe_divide(func):
    def inner(a, b):
        print("Trying to divide", a, "by", b)
        if b == 0:
            print("Division by Zero")
            return
        return func(a, b)

    return inner


@safe_divide
def divide(a, b):
    print(a / b)


divide(7, 2)
divide(5, 0)
```

Useful Python Built-In Functions A-D

- **abs(*x*)** return absolute value of *x* e.g. `abs(-5)# 5`
- **bool(*x*)** return Boolean value of *x* e.g. `bool(0) # False bool(1) # True`
- **bytearray(*[source[, encoding[, errors]]]*)** return a new array of bytes. The bytearray class is a mutable sequence of integers in the range 0 <= x < 256. e.g. `bytearray((65, 66, 66, 65))`
- **bytes(*[source[, encoding[, errors]]]*)** return a new bytes object, which is an immutable sequence of integers 0 <= x <= 255. e.g. `bytes("hello", encoding="utf-8")b'hello'`
- **chr(*i*)** return Unicode character with code point *i* e.g. `chr(8364) # €`
- **complex(*[real[, imag]*])** return a complex number with the value *real* + *imag**1j or convert a string or number to a complex number. E.g. `complex(3,-2) # 3-2j`
- **class dict(**kwarg)** return a new dictionary e.g. `dict(one=1, two=2, three=3)`
- **class dict(mapping, **kwarg)** return a new dictionary e.g. `dict({'three': 3, 'one': 1, 'two': 2})`
- **class dict(iterable, **kwarg)** return a new dictionary e.g. `dict([('two', 2), ('one', 1), ('three', 3)])`
- **dir(*[object]*)** without argument return the list of names in current scope, otherwise return list of names in given *object*

Useful Python Built-In Functions E-F

- **enumerate(*iterable, start=0*)** Return an enumerate object. Example:
```
seasons = ['Spring', 'Summer', 'Fall', 'Winter']
list(enumerate(seasons, start=1))
# [(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```
- **eval(*expression, globals=None, locals=None*)** evaluates the given *expression* and returns value. E.g. `x = 1;  result = eval('x+1')`
- **filter(*function, iterable*)** Construct an iterator from those elements of *iterable* for which *function* returns true. iterable may be either a sequence, a container which supports iteration, or an iterator. If *function* is None, the identity function is assumed, that is, all elements of *iterable* that are false are removed. E.g. `tuple(filter(lambda x>=0, (1, -4, 2, 0, -10))) # (1, 2, 0)`
- **class float([*x*])** return a floating point number constructed from a number or string ***x***. e.g. `float("1.23e3") # 1230.0`
- **format(*value[, format_spec]*)** **C**onvert a value to a "formatted" representation, as controlled by e.g. `"{:3}, {:.3f}".format(20, 2.34567) # ' 20, 2.346'`
- **class frozenset(*[iterable]*)** Return a new frozenset object, optionally with elements taken from *iterable*. `a = frozenset((1, 10, 0))`

Useful Python Built-In Functions G-L

- **globals()** return a dictionary representing the current global symbol table.
- **help([*object*])** Invoke the built-in interactive help system.
- **Input([prompt])** Wait for standard input and return it after new line. if the prompt argument is present, it is written to standard output without a trailing newline.
- **class int([*x*])**
  class int(*x*, *base*=10) Return an integer object constructed from a number or string x, or return 0 if no arguments are given. `E.g. int('1101', 2) #13`
- **isinstance(object, classinfo)** return True if object is an instance of classinfo. E.g. `isinstance('my string', str)`
- **len(s)** Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set). E.g. `len({1, 2, 100}) # 3`
- **class list([iterable])** Rather than being a function, list is actually a mutable sequence type, as documented in Lists and Sequence Types — list, tuple, range. E.g. `list(range(10,0, -1) # [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]`
- **locals()** return dictionary representing the local symbol table.

Useful Python Built-In Functions M-N

- **map(*function, iterable, ...*)** return an iterator that applies function to every item of iterable, yielding the results. E.g.
```
a = [1, 2, 100, 0, 5]
list(map(lambda x: x**3, a))
[1, 8, 1000000, 0, 125]
```
- **max(*iterable, *[, key, default]*)**
- **max(*arg1, arg2, *args[, key]*)** Return the largest item in an iterable or the largest of two or more arguments. E.g. `max(4, 0, -5) # 4`
- **min(*iterable, *[, key, default]*)**
- **min(*arg1, arg2, *args[, key]*)** Return the smallest item in an iterable or *the smallest of two or more* arguments. E.g. `min(4, 0, -5) # 5`
- **next(*iterator[, default]*)** Retrieve the next item from the iterator by calling its __next__() method.

Useful Python Built-In Functions O

- **open(*file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None*)** Open *file* and return a corresponding file object. If the file cannot be opened, an OSError is raised. Modes are:
  - 'r' open for reading (default)
  - 'w' open for writing, truncating the file first
  - 'x' open for exclusive creation, failing if the file already exists
  - 'a' open for writing, appending to the end of the file if it exists
  - 'b' binary mode
  - 't' text mode (default)
  - '+' open a disk file for updating (reading and writing)
- **ord(c)** Return an integer representing the Unicode code point of that given character (one character string). E.g. `ord('€') # returns 8364`

Useful Python Built-In Functions P-R

- **pow(x, y[, z])** return x to the power y; if z is present, return x to the power y, modulo z . E.g.
```
pow(3,5) # 243 pow(3,5,16) = 3
```
- **print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)** Print objects to the text stream file, separated by sep and followed by end. sep, end, file and flush, if present, must be given as keyword arguments. E.g.
```
print("love", "hate", sep=' & ') # love & hate
```
- **range(*stop*)**
range(*start, stop[, step=1]*) Range is actually an immutable sequence type going from 0 to *stop* or from *start* to *stop* (not including stop) with *step* stepping. E.g.
```
tuple(range(20, 0, -2))  # (20, 18, 16, 14, 12, 10, 8, 6, 4, 2)
```
- **repr(object)** return a string containing a printable representation of an object. E.g.
```
repr(3.14) # '3.14'
```
- **reversed(*seq*)** Return a reverse iterator. E.g. `a = [1, 2, 100]; list(reversed(a)) # [100, 2, 1]`
- **round(*number[, ndigits]*)** Return number rounded to ndigits precision after the decimal point. If ndigits is omitted or is None, it returns the nearest integer to its input. E.g.
```
round(123.4567, 3) # 123.457
```

Useful Python Built-In Functions S

- **class set([iterable])** Return a new set object, optionally with elements taken from iterable. E.g.
```
a=set([2, 1,2,3])
a{1, 2, 3}
b = {3, 5, 7}
c = a.union(b)
c{1, 2, 3, 5, 7}
```

- **sorted(iterable, *, key=None, reverse=False)** Return a new sorted list from the items in iterable. E.g.
```
a = (1, 100, 5, 10)
tuple(sorted(a)) # (1, 5, 10, 100)
```

- **class str(object='')**
  class str(object=b'', encoding='utf-8', errors='strict') Return a str version of object. str is the built-in string class. E.g.
```
s = b'testi' # bytes
print(str(s, encoding='utf-8')) # 'testi' string
```

- sum(iterable[, start]) Sums start and the items of an iterable from left to right and returns the total. start defaults to 0. E.g.
```
sum((1, 2, 3), 10) # 16
```

- **super([type[, object-or-type]])** Return a proxy object that delegates method calls to a parent or sibling class of type. E.g. `super().method(arg)`

Useful Python Built-In Functions T-Z

- **tuple([iterable])** Tuple is actually an immutable sequence type. E.g.
```
tuple(["one", "two", "three"]) # ('one', 'two', 'three')
```
- **class type(object)**
class type(name, bases, dict) With one argument, return the type of an object. With three arguments return a new object. E.g.
```
type("cat") # <class 'str'>
type('Student', (object,), {"name": "Jill Student", "age": 21, 'id': "13"})
```
- vars([object]) Return the __dict__ attribute for a module, class, instance, or any other object with a __dict__ attribute. Vars without arguments acts like locals(). E.g.
```
vars(jill)
```
- **zip(*iterables)** Make an iterator that aggregates elements from each of the iterables.
```
x = [1, 2, 3]
y = ['a', 'b', 'c']
list(zip(x, y)) # [(1, 'a'), (2, 'b'), (3, 'c')]
```

Regular Expressions - re module

- `import re`
- `re.search(pattern, string, flags=0)` scan the string to find first match
- `re.match(pattern, string, flags=0)` match from beginning of string
- `re.fullmatch(pattern, string, flags=0)` match entire string
- `re.split(pattern, string, maxsplit=0, flags=0)` split string by occurences of string into list of string
- `re.findall(pattern, string, flags=0)` return all non-overlapping matches of pattern in a string as a list of strings
- `re.sub(pattern, repl, string, count=0, flags=0)` replace all non-overlapping occurences of pattern in a string with repl

Using Regular Expressions

```
import re
m = re.search(r"\d{5}", "Zip code is: 12345")
m = re.match(r"^\d{5}$", "123456")
m = re.fullmatch(r"\d{5}", "12345")
print(m.string)
print(re.findall(r"\d{5}", "zip codes: 11100, 11130 and 00200"))
numbers = re.split(r", ", "1, 2, 100, -5")
s = ", ".join(numbers)
print(s)
print(re.sub(r"\d{5}", "xxxxx", "Censored: 12345 and 00200"))
```

Pre-Compiling Regular Expressions and Using Groups

```
>>> zip_re = re.compile(r"\d{5}") # faster for re-use
>>> zip_re.findall("Here are some zip codes: 11100 and 11130")
['11100', '11130']
>>> phone_re = re.compile(r"(\d{2,4})-(\d{1,10})")
>>> phone_re.match("050-1234567").groups()
('050', '1234567')
>>> phone_re.match("050-1234567").group(0)
'050-1234567'
>>> phone_re.match("050-1234567").group(1)
'050'
>>> phone_re.match("050-1234567").group(2)
'1234567'
```

Handling Files

- By default files are opened for text mode. In other words binary data, such as image files, audio/video files etc. will not work. Use "b" for binary mode.
- Basic pattern is as follows:

```
# open file mode = "r/w/a/rb/wb/ab/r+/rb+/w+/wb+/a+/ab+"
f = open("filepath", "mode")
# do your file processing d = f.read() f.write(d)
f.close() # closing file starts to flush the results
```

- But the following is even better:

```
with open("testi.txt", "mode") as f:
    # process file
    # file is closed automatically outside with
```

- If file opening fails raise OSError or subclass of it (Python 3.3 or earlier used IOError):
  - **FileNotFoundError** - If file path cannot be found
  - **PermissionError** - If lacking permissions to open file

Reading a File

- ```
  with open("test.txt") as f:
      print(f.read())
  ```

- ```
  with open("test.txt") as f:
      for line in f:
          print(line.rstrip())
  ```

- ```
  with open("test.txt") as f:
      lines = f.readlines()
      for i, line in enumerate(lines, start=1):
          print(i, line.rstrip())
  ```
  ```

Write to a File (Creates New File if Doesn't Exist)

- ```
  with open("test.txt", "w") as f:
      f.write("Hello\n2nd line\n")
  ```
- ```
  with open("test.txt", "w", encoding="utf-8") as f:
      f.writelines(["Hello ÄÖ\n", "2nd line\n"])
  ```
- ```
  with open("test.txt", "w", encoding="utf-8") as f:
      print("Hello ÄÖ", file=f)
      print("привет! 你好!", file=f)
  ```
- ```
  with open("test.txt", "w", encoding="utf-8") as f:
      while True:
          line = input("Give me a line?")
          if not line: break
          print(line, file=f)
          # print(line, file=f, flush=True)
          # alternative: you can use f.flush()
  ```

Append to a File

- Append data to the end of file:

```
with open("test.txt", "a") as f:
    f.write("one more line\n")
```

- Append data to the beginning of file:

```
with open("test.txt", "r+") as f:
    d = f.read()
    f.seek(0, 0)  # seek to start of file
    f.write("new line to start\n")
    f.write(d)  # write old contents of file
```

Reading and Copying Data from URLs

- Reading data over HTTP from URLs work just like files. Note that returned data is always binary (bytes)
- ```
  from urllib.request import urlopen
  url = https://yle.fi/uutiset
  with urlopen(url) as response:
      data = response.read().decode("utf-8")

  print(data[:1000]) # display only 1000 first characters
  ```
- ```
  from urllib.request import urlopen
  url = https://bit.ly/2T7BDAP
  with urlopen(url) as response:
      data = response.read()
      with open("cat.jpg", "wb") as f:
          f.write(data)
  ```

Using JavaScript Object Notation (JSON)

- ```
  import json
  d = {"name": "example", "number": 123, "list": [1, 2, 3]}
  with open("mydata.json", "w") as f:
      json.dump(d, f)
      print("json data saved")
  ```

- ```
  with open("mydata.json", "r") as f:
      obj = json.load(f)
      print(obj)
  ```

- ```
  d = {"name": "example ÄÖ", "number": 123, "list": [1, 2, 3]}
  # s is string in JSON format containing data from d
  s = json.dumps(d)
  print(s)
  ```

- ```
  # input string in JSON
  s = '{"list": [1, 2, 3], "number": 123, "name": "example \u00c4\u00d6"}'

  obj = json.loads(s)
  print(obj)
  ```

Serialization Using Pickle

- The pickle module implements binary protocols for serializing and de-serializing a Python object structure. “Pickling” is the process whereby a Python object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation.
- ```
  import pickle
  d = {"name": "ÄÖ", "number": 123, "list": [1, 2, 3]}
  with open("mydata.pickle", "wb") as f:
      pickle.dump(d, f)
  ```
- ```
  import pickle
  with open("mydata.pickle", "rb") as f:
      obj = pickle.load(f)
      print(obj)
  ```

Using math Library

```
import math
x = 1234.5678
print("X rounded:", round(x))
print("X rounded down:", math.floor(x))
print("X rounded up:", math.ceil(x))
print("10 base logarithm of 2 is", math.log10(2))
print("Square root of 2 is", math.sqrt(2))
print("e to power of 2 is", math.exp(2))
print("sin(0)", math.sin(0))
print("cos(0)", math.cos(0))
print("tan(0)", math.tan(0))
print("180 degrees in radians is", math.radians(180))
print("PI in degrees is", math.degrees(math.pi))
```

Pseudo-Random Numbers

- Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of 2**19937-1.

```
from random import *
print(random()) # random float between 0.0-1.0
print(randrange(1, 7)) # random int between 1..6
print(randint(1, 6)) # random int between 1..6
print(choice(["yes", "no", "maybe"])) # random choice
a = list(range(1, 101))
shuffle(a)
print(a[:10])
lottery = sample(range(1, 41), 7)
print(sorted(lottery))
```

Statistics

```
from statistics import mean, median, mode
a = [1, 0, 3, 2, 2, 10, 5, 4, 3, 2, 6, 7]
print("Average:", mean(a))
print("Median:", median(a))
print("Most common:", mode(a))

# output:
Average: 3.75
Median: 3.0
Most common: 2
```

Using Datetime Module

```
import datetime
birth = datetime.date(1990, 12, 31)
now = datetime.date.today()
print("%d.%d.%d" % (now.day, now.month, now.year))
print(now.strftime("%d.%m.%Y. Month is %B."))
delta = now-birth
print(delta)
week_in_future = now + datetime.timedelta(days=7)
delta2 = week_in_future-now
print(delta2.days)
```

Using os Module

```
import os
print("current working directory", os.getcwd())
os.chdir('/tmp')  # Change current working directory
# run system commands, which can be risky!
os.system('mkdir example123')
print(os.system('ls -la'))
print("passwd size", os.lstat('/etc/passwd').st_size)
os.system('rmdir example123')
print("HOME", os.environ["HOME"])
print("PATH", os.environ["PATH"])
f = open("test.txt", "w")
f.write("One\n")  # starts write to internal buffer
os.fsync(f.fileno())  # force changes to disk
f.close()
os.unlink("/tmp/test.txt")  # remove file
```

Using sys Module

```
import sys
print("Python version:", sys.version)
print("Python path:", sys.path)
print("Platform:", sys.platform)
print("Command line arguments:")
for arg in sys.argv[1:]:
    print(arg)
print("Standard error", file=sys.stderr)
# same as print(input())
with sys.stdin as f:
    d = f.readline()
    print(d)
sys.exit(0)
```

Enumerations

```
from enum import Enum, auto

class IceCream(Enum):
    VANILLA = 1
    CHOCOLATE = 2
    STRAWBERRY = 3
    MINT = 4

for icecream in IceCream:
    print(icecream)

class Suits(Enum):
    hearts = auto()
    spades = auto()
    clubs = auto()
    diamonds = auto()

for suit in Suits:
    print(suit)
```

Using SQLite3

```
# SQLite comes inbuilt with Python
# Good enough for testing and small usage
import sqlite3
conn = sqlite3.connect("mydatabase.db") # ":memory" for memory only
c = conn.cursor()
c.execute("CREATE TABLE IF NOT EXISTS persons (id int, name text) ")
c.execute("insert into persons values(1, 'Jack')")
c.execute("insert into persons values(2, 'Jill')")
c.execute("insert into persons values(3, 'Bob')")
conn.commit()
for id, name in c.execute("select * from persons"):
    print(id, name)

c.execute("drop table persons")  # clean up
conn.commit()
conn.close()
```

Using MySQL

```
#  first you must install and start MySQL server, create user, database, mysqlclient module

import MySQLdb
conn = MySQLdb.connect(host="localhost", user="user_name",
                       passwd="password", db="database_name")
c = conn.cursor()
c.execute("CREATE TABLE IF NOT EXISTS persons (id int, name varchar(200))")
c.execute("insert into persons values(1, 'Jack')")
c.execute("insert into persons values(2, 'Jill')")
c.execute("insert into persons values(3, 'Bob')")
conn.commit()
c.execute("select * from persons")
for id, name in c.fetchall():
    print(id, name)

c.execute("drop table persons")  # clean up
conn.commit()
c.close()
conn.close()
```

Using PostgreSQL

```
# first you must install and start PostgreSQL server, create user, database, psycopg2 module

import psycopg2
conn = psycopg2.connect(host="localhost", dbname="database_name", user="user_name",
                        password="password")
c = conn.cursor()
c.execute("CREATE TABLE IF NOT EXISTS persons" +
          "(id integer primary key, name varchar)")
c.execute("insert into persons values(1, 'Jack')")
c.execute("insert into persons values(2, 'Jill')")
c.execute("insert into persons values(3, 'Bob')")
conn.commit()
c.execute("select * from persons")
for id, name in c.fetchall():
    print(id, name)

c.execute("drop table persons") # clean up
conn.commit()
c.close()
conn.close()
```

Timing Things

```
from timeit import Timer
# manually construct a list of squares
print(Timer('a=[];\nfor x in range(1,100001): a.append(x*x)').timeit(number=5))
# use list comprehension
print(Timer('[x*x for x in range(1,100001)]').timeit(number=5))
# use generators
print(Timer('list(x*x for x in range(1,100001))').timeit(number=5))
# Multiply x by 32
print(Timer('x=13;x*32').timeit())
# Shift x 5 bits left, same as *32
print(Timer('x=13;x<<5').timeit())

# results:

0.0410432
0.028866600000000006
0.0378222
0.03077479999999999
0.031482800000000005
```

Using HTML Parser to Find All Links from a Web Page

```
from html.parser import HTMLParser
from urllib.request import urlopen
class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        if tag=="a":
            for attr, value in attrs:
                if attr=="href":
                    print("link:", value)
    #def handle_endtag(self, tag):
    #    pass
    #def handle_data(self, data):
    #    pass

parser = MyHTMLParser()
with urlopen("https://yle.fi/uutiset") as f:
    data = f.read()
    parser.feed(data.decode("utf-8"))
```

Using Least Recently Used (LRU) Cache

```
from functools import lru_cache
from urllib.request import urlopen, HTTPError
@lru_cache(maxsize=64)
def get_pep(num):
    '''Retrieve text of a Python Enhancement Proposal'''
    resource = 'http://www.python.org/dev/peps/pep-%04d/' % num
    try:
        with urlopen(resource) as s:
            return s.read().decode("utf-8")
    except HTTPError:
        return 'Not Found'

while True:
    s = input("Give me PEP number?")
    if not s: break
    if not s.isdigit(): continue
    n = int(s)
    pep = get_pep(n)
    i = pep.find("PEP ")
    j = pep.find(" |", i)
    print(n, len(pep), pep[i:j])

print(get_pep.cache_info())
```

Pip – Python Package Install Tool

- Comes with Windows Python distribution and name is pip
- Need to install separately for Linux (sudo apt install python3-pip) and name is pip3
- Install a package from Python Package Index (PyPI)
  `pip install package`
  Example
  `pip install django`
- Install a specific version:
  `pip install package==version`
  `pip install django==1.0.4`
- Uninstall package:
  `pip uninstall -y package`
  `pip uninstall -y django`
- List installed packages:
  `pip list`
- Search package by name:
  `pip search word`
  `pip search facebook`
- Show package information:
  `pip show package`
  `pip show Django`
- Update a package
  `pip install package –U`
  `pip install django -U`
- Update Pip
  `pip3 install -U pip # Linux`
  `python -m pip install -U pip# Windows`

IPython – An Improved Interactive Python Shell

- pip install ipython
- Keyboard shortcuts:
  - TAB for completion and smart indentation
  - Arrow keys up and down go through command history
  - Control-a move cursor to beginning of line
  - Control-c cancel current action
  - Control-d quit
  - Control-e move cursor to the end of line
  - Control-k delete from current cursor to end of line
  - Control-l clear screen
  - Control-u clear text on current line
  - Control-r search backwards in history
  - q quits source code reader

Useful IPython Commands

_	previous output (__ two previous, ___ three previous)
object?	Display short information about object
object??	Display long information about object
cd	change directory
ls	print directory listing
!command	run shell command
run source	run source code file
%alias name command	create alias for commands
%edit n	edit line with number n with your default editor (EDITOR environment variable in Windows) and run code
%edit filename	edit file in your default editor, %edit -p will edit previous
%quickref	print IPython documentation
%psource object	show source code for object
%pfile filename	show contents of source file
%timeit expression	time the Python expression
%% timeit setup code	time the Python expression with setup code

Virtual Environments

- Virtual environments allow you to use different versions of libraries.

1. Install virtualenv tool
   ```
   pip3 install virtualenv
   ```
2. Create a new virtual environment
   ```
   cd project_folder
   virtualenv venv # instead of venv, you can use other names e.g. env
   ```
3. Activate virtual environment
   ```
   source venv/bin/activate # Linux
   venv/venv/activate.bat   # Windows
   ```
4. Install packages like usual
   ```
   pip3 install package
   ```
5. Deactivate
   ```
   deactivate
   ```
6. Delete virtual environment
   ```
   rm -rf venv
   ```

Using Requirements

- How to create a requirements.txt file, which describes what Python libraries (and their versions) are needed for a project. Preferably you do this inside a virtual environment.

```
% pip3 freeze > requirements.txt
% cat requirements
asgiref==3.3.1
Django==3.1.5
Pillow==8.1.0
pytz==2020.5
sqlparse==0.4.1
% pip3 install -r requirements.txt
```

Using pipenv

- Pipenv is a higher level user-interface for virtualenv
- `pip3 install --user pipenv`
- `cd project_folder`
- `pipenv install` *`package`*
- `pipenv run python3 main.py`
- `pipenv shell`
- `deactivate`

Virtual Environment Tools

- You can also use the built-in venv module:
  ```
  python3 -m venv /path/to/new/virt_environment
  ```
- Virtualenvwrapper makes using virtual environments easier:
  - `mkvirtualenv envname  # create new environment`
  - `workon project_folder # work on project`
  - `lsvirtualenv          # list all virtual envs`
  - `cdvirtualenv     # navigate to current virtual env`
  - `cdsitepackages   # cd into site-packages directory`
  - `lssitepackages   # list site-packages directory`

Python Imaging Library (PIL)

- A free open-source library, initially released in 1995.
- Supports loading and saving into lots of different image file formats
- Per-pixel manipulations, masking, transparency
- Image filtering, such as blurring or edge finding
- Image enhancing, such as sharpening or adjusting brithgness, contrast
- Add text to images
- Drawing
- newer versions known as Pillow

Using PIL / Pillow

```
# Convert image into different formats
from PIL import Image
im = Image.open("pic.gif") # open a GIF file
im.save("pic.png") # PNG supports both P and RGB mode
im2 = im.convert("RGB") # convert P mode image to RGB mode
im2.save("pic.jpg", quality=90, optimize=True)

# resize image while retaining aspect ratio
from PIL import Image
im = Image.open("../assets/fitness.jpg")
width, height = im.size
ratio = width/height
new_width = 640
new_height = new_width/ratio
im2 = im.resize((round(new_width), round(new_height)), Image.ANTIALIAS)
im2.save("fitness2.jpg", quality=95, optimize=True)
```

Zen of Python (by Tim Peters)

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one—and preferably only one—obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than right now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea—let's do more of those!