Python – DI2006(P), DI2006(F)

Pablo Picazo-Sanchez

September 2022

1 Simple Data Types

P1.1 Write a program that prints the string `"Hello World!"`.

P1.2 Write a program that stores the string `"Hello World!"` into a variable and later prints it.

P1.3 Write a program that prints the string `"Hello <name>!"`, where `<name>` is the name introduced by the user.

P1.4 Write a program that outputs the result of the following operation: $\left(\frac{3+2}{2\cdot 5}\right)^2$

P1.5 Write a program that asks the user for the number of working hours (a week) and the cost per hour. The program outputs the monthly salary.

P1.6 Write a program that given n, a positive number ($\mathbb{N}$), outputs the sum of all the numbers until n. The general formula is: $\frac{n(n+1)}{2}$

P1.7 Write a program that given the weight (kg) and height (meters), calculates and stores the Body Mass Index (BMI) into a variable. Later, it outputs the string: `"Your BMI is <bmi>"`, where `<bmi>` is the BMI previously computed with exactly 2 decimals.

P1.8 Write a program that given two numbers $n, m \in \mathbb{N}$, outputs the following string: `"<n> divided by <m> gives a quotient of <q> with a remainder of <r>"` where `"<n>"` and `"<m>"` are the numbers input by the user and `"<q>"` and `"<r>"` are the quotient and the remainder of the division, respectively.

P1.9 Write a program to compute the future value of a specified principal amount, rate of interest, and a number of years.

P1.10 An online toy shop is becoming popular because of two main items: dolls and cards. To deliver the items, the delivery company charges the toy shop based on the weight of the items (10SEK/gram). Knowing that a doll weights 112g and cards 75g, write a program that computes the total amount before the toy shop sends the package.

P1.11 Imagine our bank offers us a rate of interest of 4%/year. These interests are not added to our account until next year. Write a program that given an amount (savings), computes the interests after the first, second, and third year. Truncate the output to 2 decimals.

P1.12 A bakery sells cakes at 140 SEK. When a cake is not sold, the next day that cake has a 60% discount. Write a program that given the an amount of not-sold cakes it outputs the usual price and the final price (with the discount for not being sold).

2 Strings

P2.1   Write a program that asks for the user name and a number $n \in \mathbb{N}$ and prints that username n times (in different lines).

P2.2   Write a program that given the first, middle and last name, it outputs the name (in different lines): 1) in lowercase; 2) in uppercase, and; 3) capitalized.

P2.3   Write a program that given a name, it outputs: `<NAME> has <n> letters` where `<NAME>` is the input name (uppercase) and `<n>` the number of letters the name has.

P2.4   Phone numbers usually follow *prefix-number-extension* pattern, where prefix is the country code (e.g., +34, +46), the number has a variable amount of digits, and the extension is the last two numbers. Write a program that given a number with that format, it outputs the phone number without the prefix nor the extension.

P2.5   Write a program that given a sentence, it outputs the inverted sentence.

P2.6   Write a program that given a sentence and a vowel, it replaces that vowel in the sentence by the uppercase.

P2.7   Write a program that given an email, it replaces the domain (string after @) by hh.se.

P2.8   Write a program that given an amount (in €) with two decimals, it outputs the amount of € and cents.

P2.9   Given a date with the *dd/mm/aaaa* format, write a program that splits and outputs the day, month and year of the date.

P2.10   Write a program that given a list of items split by the comma character (,), it outputs every item in different lines (No loops!).

P2.11   Given an item, the price (SEK) and the number of items, it outputs the name of the product, its price (9 digits and 2 decimals), number of items, and the total cost (8 digits and 2 decimals).

3 Conditions

P3.1 Write a program that given an age, it outputs whether it is an adult or not.

P3.2 Write a program that given a string already stored, it outputs whether the user guessed it correctly or not (not case sensitive).

P3.3 Write a program that given two numbers $n, m \in \mathbb{N}$ it outputs $\frac{n}{m}$ (if $m == 0$ then it outputs an error).

P3.4 Write a program that given a number $n \in \mathbb{N}$ it outputs whether it is odd or even.

P3.5 To declare taxes, you need to be 16 and your salary should be higher than 10,000SEK. Write a program that given the age and the salary, it outputs whether the user has to declare taxes or not.

P3.6 In a particular course, teachers have split students into two groups A and B according to the sex and the name. Group A is composed of women whose name is before M and men whose name if after N. Group B is composed of all the rest. Write a program that given a name and the sex, it outputs which group the user belongs to.

P3.7 A school has following rules for grading system:

Below 25:	F
25 to 45:	E
45 to 50:	D
50 to 60:	C
60 to 80:	B
Above 80:	A

Ask user to enter marks and print the corresponding grade.

P3.8 Write a program to print the absolute value of a number entered by user (e.g., INPUT: 1 OUTPUT: 1; INPUT: -1 OUTPUT: 1)

P3.9 The prices of a restaurant are calculated according to the age of the customers. Children under 4 do not pay. Children from 4 to 18, they pay 150SEK, while adults pay 180SEK. Write a program that given the age, it outputs the price.

P3.10 An Italian restaurant has 5 pizzas classified into vegetarian and not vegetarian pizzas. The vegetarian pizzas are made of peppers or artichokes. For not vegetarian pizzas, they have pepperoni, ham, and salmon. Write a program that asks users for the pizza they want.

P3.11 Fix the following code:

```python
pass = input('Pass: ")
if pass in ['password'):
  print('OK')
else
  print('ERROR')
```

4 Loops

P4.1 Write a program that given a word, it prints 10 times that word.

P4.2 Convert the following loop into a for loop:

```
x = 4
while (x<=10):
    print(x)
    x += 2
```

P4.3 Write a program that given a number, it outputs all the odd numbers from 1 to that number split by commas (,).

P4.4 Write a countdown from a given number until 0 split by commas (,).

P4.5 Given an amount, number of years, and rate interest, write a program that outputs the total amount every year.

P4.6 Write a program that given a number, it outputs a triangle like the following (n=5):
```
*
**
***
****
*****
```

P4.7 Write a program that prints the multiplication table from 1 to 10.

P4.8 Write a program that prints numbers from 1 to 20 except multiple of 2 and 3.

P4.9 Write a program that given a string already stored, it outputs whether the user guessed it correctly or not (not case sensitive) until the user guesses it.

P4.10 Write a program that checks whether a number is prime or not.

P4.11 Write a program that given a word, it reverses the word.

P4.12 Given a sentence and a word, write a program that counts the number of times that word appears in the sentence.

P4.13 Write a program that echoes what the user inputs until the string `"end"` is found.

5   List & Tuples

P5.1   Write a program that prints a list.

P5.2   Write a program that prints every element of a list in different lines.

P5.3   Write a program that given 4 courses, it asks the user for the marks on every course and finally prints the course together with the mark.

P5.4   Write a program that given a list of numbers split by commas (,) it sorts (ascending order) and outputs that list.

P5.5   Write a program that prints a list of numbers from 1..10 in reverse order split by commas (,).

P5.6   We say that a course is passed if the mark is at least 5 (up to 10). Write a program that given 4 courses, it asks the user for the marks on every course and removes the ones the student passed.

P5.7   Given a list with all the letters of the alphabet, write a program the removes those letters whose position is multiple of 3.

P5.8   Write a program that checks whether a word is palindrome or not (no loops).

P5.9   Write a program that given a sentence, counts the number of appearance of every vowel in the sentence.

P5.10   Given a list of numbers, write a program that prints the lowest and the largest numbers.

P5.11   Given two vectors a=(1,2,3) and b=(-1,0,2) write a number that outputs the scalar product (dot product).

P5.12   Write a program that generates a Fibonacci sequence.

P5.13   Given a list of numbers, write a program that prints the mean ($\mu = \frac{1}{n} \sum_{i=i}^{n} x_i$), the variance ($\sigma^2 = \sum_{i=i}^{n} (x_i - \mu)^2$), and the standard deviation ($\sigma = \sqrt{\sigma^2}$).

6   Dictionaries

P6.1   Write a program that given a sentence, counts the number of appearance of every vowel in the sentence and stores it in a dictionary.

P6.2   Write a program that stores the name, age, address, and phone number of a user and stores it into a dictionary. Later it has to print "`<name> is <age> years old, lives in <address> and the phone number is <phone>`".

P6.3   Write a program that stores all the values of the table above. Later, given a fruit and the weight (kg), it outputs the price of the purchase.

Fruit	Price
Banana	1.35
Apple	0.80
Pear	0.85
Orange	0.70

P6.4   Write a program that given a date (dd/mm/aaaa) it replaces the month by its name and shows the string: "`The <dd> of <month_name>, <aaaa>`"

P6.5   Write a program that given a dictionary (e.g., `{'Python': 6, 'JavaScript': 4, 'Pascal': 5}`) prints every subject together with the number of credits. Finally it should also show the sum of all the credits.

P6.6   Write a program that stores elements into a dictionary until the user types `'N'`.

P6.7   Write a program that converts two lists (same length) into a dictionary

P6.8   Write a program that merges two dictionaries into one.

P6.9   Given the following dictionary:

```
sampleDict = {
    "class": {
        "student": {
            "name": "Mike",
            "marks": {
                "physics": 70,
                "history": 80
            }
        }
    }
```

```
}
```

Write a program that prints the value of `'history'` key.

6.10 Fix the following code:

```
agenda = {'Oliver':123456789, 'Erik':987654321}

def delete(agenda, user):
    del(agenda[user])
    return agenda[user]

print(delete(agenda, 'Ana'))
```

7 Sets

P7.1   Given a Python list, write a program to add all its elements into a given set.

P7.2   Write a program that returns a new set of identical items from two sets `set1 = 10, 20, 30, 40, 50` and `set2 = 30, 40, 50, 60, 70`

P7.3   Write a Python program to return a new set with unique items from `set1 = 10, 20, 30, 40, 50` and `set2 = 30, 40, 50, 60, 70` sets.

P7.4   Given two Python sets (`set1 = 10, 20, 30, 40, 50` and `set2 = 30, 40, 50, 60, 70`), write a program to update the first set with items that exist only in the first set and not in the second set.

P7.5   Given two Python sets (`set1 = 10, 20, 30, 40, 50` and `set2 = 30, 40, 50, 60, 70`), write a program that returns a set of elements present in set1 or set2, but not both.

P7.6   Given two Python sets (`set1 = 10, 20, 30, 40, 50` and `set2 = 30, 40, 50, 60, 70`), write a program that checks if they have any elements in common. If yes, display the common elements

8 Functions

P8.1 Write a function that outputs `'Hello World!'`.

P8.2 Write a program to create a function that takes two arguments, name and age, and print their value.

P8.3 Write a program that prints the factorial $(n! = 1 \cdot 2 \ldots \cdot (n-1) \cdot n)$ of a number n.

P8.4 Write a program to create function `func1()` to accept a variable length of arguments and print their value.

P8.5 Write a program that calculates the area of a circle $(\pi \cdot r^2)$ as well as the volume of a cylinder $\pi \cdot r^2 \cdot h$.

P8.6 Write a function that return the mean of a list of numbers.

P8.7 Write a function that given a list of numbers, it returns another list with the square of the numbers.

P8.8 Write a function that given a list of numbers, it returns a dictionary with the mean, variance and standard deviation of the list of numbers.

P8.9 Write a function that computes the Greatest Common Divisor (GCD) and another one that computes the Least Common Multiple (LCM).

P8.10 Write a function that converts a decimal number into binary and another one that converts a binary number into decimal.

P8.11 Write a function that given two numbers $n, m \in \mathbb{R}$ it returns the addition and subtraction.

P8.12 Write a function that given two inputs, the employee's name and salary, it displays both. Also, if the salary is missing in the function call then assign default value 9000 to salary.

P8.13 Write a function that allows computing sine (sin), cosine (cos), tangent (tan), exponential (exp), and logarithmic (log) functions. The program will apply such function to a range of numbers, going from 1..n, being n a number given by the user.
HINT: `from math import sin, cos, tan, exp, log`

P8.14 Write a function that receives a function and a list and returns a list with the elements of the input list after applying the function.

P8.15 Write a function that receives a Boolean function and a list and returns a list with the elements of the input list that satisfies condition of the function.

P8.16 Write a function that given a string, it returns a dictionary with the letters and the frequency.

P8.17 Write a function that given a dictionary of courses and grades, it outputs another dictionary with the courses in capital letters and the qualifications of the passed courses based according to the following table:

Grade	Qualifications
<5	F
<6	E
<7	D
<8	C
<9	B
<=10	A

HINT: Use `map()` function that returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple etc.).

P8.18 Write a function that computes the module of a vector. **HINT:** Use `from functools import reduce`. This function performs a rolling-computation as specified by the passed function to the neighboring elements, by taking a function and an iterable as arguments, and returns the final computed value.

P8.19 Write a function that given a list of number, it returns a list with only even numbers. **HINT:** use `filter()` function, a function that extracts elements from an iterable (list, tuple etc.) for which a function returns True.

9 Files

P9.1 Write a program that given a number $n \in [1, .., 10]$, it creates a file named table_n.txt with the multiplication table of n.

P9.2 Write a program that given a number $n \in [1, .., 10]$, it reads a file named table_n.txt and prints the result. If the file does not exist, it should handle the error.

P9.3 Write a program that given two numbers $n, m \in [1, .., 10]$, it reads a file named table_n.txt and prints the line m as a result. If the file does not exist, it should handle the error.

P9.4 Write a function that given the URL of a text file, it shows the number of words of that file. **HINT:** use the `from urllib import request` function to open the URL which can be either a string or a Request object.

P9.5 Write a program that opens the a file with the GDP of every country in the European Union (https://ec.europa.eu/eurostat/estat-navtree-portlet-prod/BulkDownloadListing?file=data/sdg_08_10.tsv.gz&unzip=true), and given a country code, it shows the GDP of all the years available.

P9.6 Write a program that manages the agenda (clients and phone numbers) of a company. The program should allow the company to query, create, and delete items from the agenda. The agenda should be stored in a file called agenda.txt where users and phones should be split by comas (,) and every entry should be in different lines of the file.

P9.7 Convert the following dictionary: `data = "key1": "value1", "key2": "value2"` into JSON format.

P9.8 Access the value of key2 from the following JSON: `key1": "value1", "key2": "value2"sampleJson = """"""`

P9.9 PrettyPrint following JSON `sampleJson = "key1": "value1", "key2": "value2"` with indent level 2 and key-value separators should be (`","`, `" = "`).

P9.10 Sort JSON keys in and write them into a file.

10 Classes

P10.1   Create a Vehicle class without any variables and methods

P10.2   Write a Python program to create a Vehicle class with `max_speed` and `mileage` instance attributes.

P10.3   Create a child class Bus that will inherit all of the variables and methods of the Vehicle class

P10.4   Create a Bus class that inherits from the Vehicle class. Give the capacity argument of `Bus.seating_capacity()` a default value of 50.

Use the following code for your parent Vehicle class.

```python
class Vehicle:
    def __init__(self, name, max_speed, mileage):
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage

    def seating_capacity(self, capacity):
        return f"The seating capacity of a {self.name} is {
            capacity} passengers"
```

P10.5   Define a class attribute *color* with a default value *white*, i.e., Every Vehicle should be white.

```python
class Vehicle:

    def __init__(self, name, max_speed, mileage):
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage

class Bus(Vehicle):
    pass

class Car(Vehicle):
    pass
```

P10.6   Convert the following Vehicle Object into JSON

```python
class Vehicle:
    def __init__(self, name, engine, price):
        self.name = name
        self.engine = engine
```

```
        self.price = price

vehicle = Vehicle("Toyota Rav4", "2.5L", 32000)
```

P10.7 Convert the previous JSON into a Vehicle Object

Python 3 Cheat Sheet

Base Types

integer, float, boolean, string, bytes

int `783  0  -192`   `0b010` *binary*   `0o642` *octal*   `0xF3` *hexa*
            zero

float `9.23  0.0   -1.7e-6`
                              ×10⁻⁶ → $\times 10^{-6}$

bool `True  False`

str `"One\nTwo"`   *Multiline string:*
         escaped new line   `"""X\tY\tZ`
`'I\'m'`   `1\t2\t3"""`
   escaped '   *escaped tab*

bytes `b"toto\xfe\775"`
                hexadecimal octal   ☞ *immutables*

Container Types

■ **ordered sequences**, fast index access, repeatable values

list `[1,5,9]`   `["x",11,8.9]`   `["mot"]`   `[]`

tuple `(1,5,9)`   `11,"y",7.4`   `("mot",)`   `()`

Non modifiable values (immutables)   ☞ *expression with only comas →* ***tuple***

str bytes *(ordered sequences of chars / bytes)*   `""`   `b""`

■ **key containers**, no *a priori* order, fast key access, each key is unique

dictionary **dict** `{"key":"value"}`   `dict(a=3,b=4,k="v")`   `{}`
(key/value associations) `{1:"one",3:"three",2:"two",3.14:"π"}`

collection **set** `{"key1","key2"}`   `{1,9,3,0}`   `set()`
☞ keys=hashable values (base types, immutables…) **frozenset** *immutable set*   *empty*

Identifiers

for variables, functions, modules, classes… names

a…zA…Z_ followed by **a…zA…Z_0…9**
□ diacritics allowed but should be avoided
□ language keywords forbidden
□ lower/UPPER case discrimination
   ☺ `a toto x7 y_max BigOne`
   ☹ ~~8y and for~~

Variables assignment

=

☞ assignment ⇔ **binding** of a *name* with a *value*
1) evaluation of right side expression value
2) assignment in order with left side names

`x=1.2+8+sin(y)`
`a=b=c=0` *assignment to same value*
`y,z,r=9.2,-7.6,0` *multiple assignments*
`a,b=b,a` *values swap*
`a,*b=seq` } *unpacking of sequence in*
`*a,b=seq` } *item and list*
`x+=3` *increment ⇔* `x=x+3`   *and*
`x-=2` *decrement ⇔* `x=x-2`   `*=`
`x=None` *« undefined » constant value*   `/=`
`del x` *remove name x*   `%=`
                                        …

Conversions

type (*expression*)

`int("15")  → 15`
`int("3f",16)  → 63`   *can specify integer number base in 2ⁿᵈ parameter*
`int(15.56)  → 15`   *truncate decimal part*
`float("-11.24e8")  → -1124000000.0`
`round(15.56,1)→15.6`   *rounding to 1 decimal (0 decimal → integer number)*
`bool(x)` `False` for null **x**, empty container **x** , **None** or **False x** ; **True** for other **x**
`str(x)→ "…"`   *representation string of* **x** *for display (cf. formatting on the back)*
`chr(64)→'@'  ord('@')→64`   *code ↔ char*
`repr(x)→ "…"`   *literal representation string of* **x**
`bytes([72,9,64])  → b'H\t@'`
`list("abc")  → ['a','b','c']`
`dict([(3,"three"),(1,"one")])  → {1:'one',3:'three'}`
`set(["one","two"])  → {'one','two'}`
separator **str** *and sequence of* **str** *→ assembled* **str**
   `':'.join(['toto','12','pswd'])  → 'toto:12:pswd'`
str *splitted on whitespaces →* **list** *of* **str**
   `"words with  spaces".split()  → ['words','with','spaces']`
str *splitted on separator* **str** *→* **list** *of* **str**
   `"1,4,8,2".split(",")  → ['1','4','8','2']`
sequence of one type → **list** *of another type (via list comprehension)*
   `[int(x) for x in ('1','29','-3')]  → [1,29,-3]`

Sequence Containers Indexing

for lists, tuples, strings, bytes…

	-5	-4	-3	-2	-1
negative index					
positive index	0	1	2	3	4
`lst=[10,`	`20,`	`30,`	`40,`	`50]`	
positive slice	0	1	2	3	4	5
negative slice	-5	-4	-3	-2	-1	

Items count
`len(lst)→5`
☞ **index from 0**
(here from 0 to 4)

Individual access to **items** via `lst[`*index*`]`
`lst[0]→10`   ⇒ *first one*   `lst[1]→20`
`lst[-1]→50`   ⇒ *last one*   `lst[-2]→40`

*On mutable sequences (**list**), remove with*
`del lst[3]` *and modify with assignment*
`lst[4]=25`

Access to **sub-sequences** via `lst[`*start slice* : *end slice* : *step*`]`

`lst[:-1]→[10,20,30,40]`   `lst[::-1]→[50,40,30,20,10]`   `lst[1:3]→[20,30]`   `lst[:3]→[10,20,30]`
`lst[1:-1]→[20,30,40]`   `lst[::-2]→[50,30,10]`   `lst[-3:-1]→[30,40]`   `lst[3:]→[40,50]`
`lst[::2]→[10,30,50]`   `lst[:]→[10,20,30,40,50]` *shallow copy of sequence*

Missing slice indication → from start / up to end.
*On mutable sequences (**list**), remove with* `del lst[3:5]` *and modify with assignment* `lst[1:4]=[15,25]`

Boolean Logic

Comparisons : `< > <= >= == !=`
(boolean results)   ≤ ≥ = ≠

a and b logical and *both simulta-*
                        -neously

a or b logical or *one or other*
                      or both

☞ pitfall : **and** *and* **or** *return* <u>value</u> *of* **a** *or*
of **b** *(under shortcut evaluation).*
⇒ *ensure that* **a** *and* **b** *are booleans.*

not a logical not

True
False   True and False constants

Statements Blocks

parent statement **:**
→ *statement block 1…*
       ⋮
parent statement **:**
       statement block2…
              ⋮
next statement after block 1

indentation !

☞ *configure editor to insert 4 spaces in place of an indentation tab.*

Modules/Names Imports

module **truc**⇔*file* ***truc.py***
`from monmod import nom1,nom2 as fct`
                    →*direct access to names, renaming with* **as**
`import monmod` →*access via* `monmod.nom1` …
☞ *modules and packages searched in* python path *(cf* `sys.path`*)*

Conditional Statement

statement block executed only
if *a condition is true*

   if *logical condition* **:**
   ──┤ *statements block*

Can go with several *elif*, *elif*… and only one
final *else*. Only the block of first true
condition is executed.

☞ *with a var* **x**:
`if bool(x)==True:` ⇔ `if x:`
`if bool(x)==False:`⇔ `if not x:`

`if age<=18:`
    `state="Kid"`
`elif age>65:`
    `state="Retired"`
`else:`
    `state="Active"`

Maths

☞ *floating numbers… approximated values*

Operators: `+ - * / // % **`
Priority `(…)`   × ÷   ↑   ↗   aᵇ
              integer ÷   ÷ remainder

`@` → matrix × *python3.5+numpy*

`(1+5.3)*2→12.6`
`abs(-3.2)→3.2`
`round(3.57,1)→3.6`
`pow(4,3)→64.0`
☞ *usual order of operations*

angles in radians
`from math import sin,pi…`
`sin(pi/4)→0.707…`
`cos(2*pi/3)→-0.4999…`
`sqrt(81)→9.0` √
`log(e**2)→2.0`
`ceil(12.5)→13`
`floor(12.5)→12`
modules **math**, **statistics**, **random**,
decimal, **fractions**, **numpy**, *etc. (cf. doc)*

Exceptions on Errors

Signaling an error:
   raise *ExcClass(…)*
Errors processing:
   try:
   ──┤ *normal procesising block*
   except *Exception* **as e:**
   ──┤ *error processing block*

☞ **finally** *block for final processing in all cases.*

Conditional Loop Statement

statements block executed **as long as** *condition is true*

while *logical condition* **:**
→ *statements block*

beware of infinite loops!

```
s = 0          initializations before the loop
i = 1          condition with a least one variable value (here i)
while i <= 100:
    s = s + i**2
    i = i + 1          make condition variable change !
print("sum:",s)
```

Loop Control

break	*immediate exit*
continue	*next iteration*
else block for *normal* *loop exit.*	

Algo:
$$s = \sum_{i=1}^{i=100} i^2$$

Iterative Loop Statement

statements block executed **for each** *item of a container or iterator*

for *var* **in** *sequence* **:**
→ *statements block*

good habit : don't modify loop variable

Go over sequence's **values**
```
s = "Some text"      initializations before the loop
cnt = 0
              loop variable, assignment managed by for statement
for c in s:
    if c == "e":
        cnt = cnt + 1
    print("found",cnt,"'e'")
```
Algo: count number of e *in the string.*

loop on dict/set ⇔ loop on keys sequences
use *slices* to loop on a subset of a sequence

Go over sequence's **index**
□ modify item at index
□ access items around index (before / after)
```
lst = [11,18,9,12,23,4,17]
lost = []
for idx in range(len(lst)):
    val = lst[idx]
    if val > 15:
        lost.append(val)
        lst[idx] = 15
print("modif:",lst,"-lost:",lost)
```
Algo: limit values greater than 15, memorizing of lost values.

Go simultaneously over sequence's **index** and **values**:
```
for idx,val in enumerate(lst):
```

Display

```
print("v=",3,"cm :",x,",",y+4)
```
items to display : literal values, variables, expressions

print options:
□ **sep=" "**   items separator, default space
□ **end="\n"**   end of print, default new line
□ **file=sys.stdout** print to file, default standard output

Input

```
s = input("Instructions:")
```
input always returns a **string**, convert it to required type
(cf. boxed *Conversions* on the other side).

Generic Operations on Containers

len(c) → items count
min(c) max(c) sum(c)
sorted(c) → **list** sorted *copy*
val in c → boolean, membership operator **in** (absence **not in**)
enumerate(c) → *iterator* on (index, value)
zip(c1,c2…) → *iterator* on tuples containing c_i items at same index
all(c) → **True** if **all** c items evaluated to true, else **False**
any(c) → **True** if **at least one** item of c evaluated true, else **False**

Note: For dictionaries and sets, these operations use **keys**.

Specific to ***ordered sequences containers*** *(lists, tuples, strings, bytes…)*
reversed(c) → *inversed iterator*   **c*5** → duplicate   **c+c2** → concatenate
c.index(val) → *position*   **c.count(val)** → *events count*
import copy
copy.copy(c) → shallow copy of container
copy.deepcopy(c) → deep copy of container

Integer Sequences

range([start,] end [,step])
start default 0, *end* not included in sequence, *step* signed, default 1
range(5) → 0 1 2 3 4   **range(2,12,3)** → 2 5 8 11
range(3,8) → 3 4 5 6 7   **range(20,5,-5)** → 20 15 10
range(len(seq)) → *sequence of index of values in seq*
range provides an immutable sequence of int constructed as needed

Operations on Lists

modify original list
lst.append(val)   add item at end
lst.extend(seq)   add sequence of items at end
lst.insert(idx,val)   insert item at index
lst.remove(val)   remove first item with value *val*
lst.pop([idx]) → *value* remove & return item at index *idx* (default last)
lst.sort()   lst.reverse() sort / reverse liste *in place*

Function Definition

function name (identifier)
        named parameters

```
def fct(x,y,z):
    """documentation"""
    # statements block, res computation, etc.
    return res  ← result value of the call, if no computed
                   result to return: return None
```

[fct]

parameters and all
variables of this block exist only *in the block* and *during the function call* (think of a "black box")

Advanced: **def fct(x,y,z,*args,a=3,b=5,**kwargs):**
args variable positional arguments (→ **tuple**), default values,
kwargs variable named arguments (→* **dict)

Function Call

```
r = fct(3,i+2,2*i)
```
storage/use of returned value   *one argument per parameter*

this is the use of function name with parentheses *which does the call*

Advanced:
*sequence
**dict

[fct()] → [fct]

Operations on Dictionaries

d[key]=value   **d.clear()**
d[key] → *value*   **del d[key]**
d.update(d2) ⎱ update/add associations
d.keys() ⎱
d.values() ⎭ → *iterable views on keys/values/associations*
d.items()
d.pop(key[,default]) → *value*
d.popitem() → *(key,value)*
d.get(key[,default]) → *value*
d.setdefault(key[,default]) → *value*

Operations on Sets

Operators:
| → union (vertical bar char)
& → intersection
− ^ → difference/symmetric diff.
< <= > >= → inclusion relations
Operators also exist as methods.

s.update(s2)   s.copy()
s.add(key)   s.remove(key)
s.discard(key)   s.clear()
s.pop()

Files

storing data on disk, and reading it back
```
f = open("file.txt","w",encoding="utf8")
```
file **variable** for operations
name of file on disk (+path…)
opening **mode**
□ **'r'** read
□ **'w'** write
□ **'a'** append
□ …**'+' 'x' 'b' 't'**

encoding of chars for *text files*:
utf8   ascii
latin1   …

cf. modules **os**, **os.path** and **pathlib**

writing
f.write("coucou")
f.writelines(list of lines)

read empty string if end of file   **reading**
f.read([n])   → *next chars*
if n not specified, read up to end !
f.readlines([n])   → **list** of next lines
f.readline()   → *next line*
text mode **t** *by default (read/write* **str**), *possible binary mode* **b** *(read/write* **bytes**). **Convert from/to required type !**

f.close()   *dont forget to* **close the file** *after use !*

f.flush() write cache   **f.truncate([size])** resize
reading/writing progress sequentially in the file, modifiable with:
f.tell() → *position*   **f.seek(position[,origin])**

Very common: opening with a guarded block
(automatic closing) and reading loop on lines
of a text file:
```
with open(…) as f:
    for line in f :
        # processing of line
```

Operations on Strings

s.startswith(prefix[,start[,end]])
s.endswith(suffix[,start[,end]]) s.strip([chars])
s.count(sub[,start[,end]]) s.partition(sep) → *(before,sep,after)*
s.index(sub[,start[,end]]) s.find(sub[,start[,end]])
s.is…() *tests on chars categories (ex.* **s.isalpha()**)
s.upper()   s.lower()   s.title()   s.swapcase()
s.casefold()   s.capitalize()   s.center([width,fill])
s.ljust([width,fill])   s.rjust([width,fill])   s.zfill([width])
s.encode(encoding)   s.split([sep])   s.join(seq)

Formatting

formating directives   values to format
```
"modele{} {} {}".format(x,y,r) ⟶ str
```
"{selection:formatting!conversion}"

□ **Selection** :
    2
    nom
    0.nom
    4[key]
    0[2]

Examples:
"{:+2.3f}".format(45.72793)
→'+45.728'
"{1:>10s}".format(8,"toto")
→'   toto'
"{x!r}".format(x="I'm")
→'"I\'m"'

□ **Formatting** :
fill char alignment sign mini width . precision~maxwidth type
< > ^ =   + − space   0 at start for filling with 0
integer: **b** binary, **c** char, **d** decimal (default), **o** octal, **x** or **X** hexa…
float: **e** or **E** exponential, **f** or **F** fixed point, **g** or **G** appropriate (default),
string: **s** …   **%** percent
□ **Conversion** : **s** (readable text) or **r** (literal representation)

Python For Data Science *Cheat Sheet*

Python Basics

Learn More Python for Data Science Interactively at www.datacamp.com

Variables and Data Types

Variable Assignment

```
>>> x=5
>>> x
 5
```

Calculations With Variables

`>>> x+2`		Sum of two variables
` 7`		
`>>> x-2`		Subtraction of two variables
` 3`		
`>>> x*2`		Multiplication of two variables
` 10`		
`>>> x**2`		Exponentiation of a variable
` 25`		
`>>> x%2`		Remainder of a variable
` 1`		
`>>> x/float(2)`		Division of a variable
` 2.5`		

Types and Type Conversion

`str()`	`'5'`, `'3.45'`, `'True'`	Variables to strings
`int()`	`5`, `3`, `1`	Variables to integers
`float()`	`5.0`, `1.0`	Variables to floats
`bool()`	`True`, `True`, `True`	Variables to booleans

Asking For Help

```
>>> help(str)
```

Strings

```
>>> my_string = 'thisStringIsAwesome'
>>> my_string
'thisStringIsAwesome'
```

String Operations

```
>>> my_string * 2
 'thisStringIsAwesomethisStringIsAwesome'
>>> my_string + 'Innit'
 'thisStringIsAwesomeInnit'
>>> 'm' in my_string
 True
```

Lists
Also see NumPy Arrays

```
>>> a = 'is'
>>> b = 'nice'
>>> my_list = ['my', 'list', a, b]
>>> my_list2 = [[4,5,6,7], [3,4,5,6]]
```

Selecting List Elements
Index starts at 0

Subset
`>>> my_list[1]`	Select item at index 1
`>>> my_list[-3]`	Select 3rd last item

Slice
`>>> my_list[1:3]`	Select items at index 1 and 2
`>>> my_list[1:]`	Select items after index 0
`>>> my_list[:3]`	Select items before index 3
`>>> my_list[:]`	Copy my_list

Subset Lists of Lists
`>>> my_list2[1][0]`	`my_list[list][itemOfList]`
`>>> my_list2[1][:2]`	

List Operations

```
>>> my_list + my_list
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list * 2
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list2 > 4
True
```

List Methods

`>>> my_list.index(a)`	Get the index of an item
`>>> my_list.count(a)`	Count an item
`>>> my_list.append('!')`	Append an item at a time
`>>> my_list.remove('!')`	Remove an item
`>>> del(my_list[0:1])`	Remove an item
`>>> my_list.reverse()`	Reverse the list
`>>> my_list.extend('!')`	Append an item
`>>> my_list.pop(-1)`	Remove an item
`>>> my_list.insert(0,'!')`	Insert an item
`>>> my_list.sort()`	Sort the list

String Operations
Index starts at 0

```
>>> my_string[3]
>>> my_string[4:9]
```
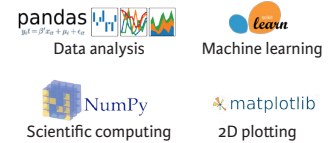
String Methods

`>>> my_string.upper()`	String to uppercase
`>>> my_string.lower()`	String to lowercase
`>>> my_string.count('w')`	Count String elements
`>>> my_string.replace('e', 'i')`	Replace String elements
`>>> my_string.strip()`	Strip whitespaces

Libraries

Import libraries
```
>>> import numpy
>>> import numpy as np
```
pandas Data analysis
learn Machine learning

Selective import
```
>>> from math import pi
```
NumPy Scientific computing
matplotlib 2D plotting

Install Python

ANACONDA Leading open data science platform powered by Python

spyder Free IDE that is included with Anaconda

jupyter Create and share documents with live code, visualizations, text, ...

Numpy Arrays
Also see Lists

```
>>> my_list = [1, 2, 3, 4]
>>> my_array = np.array(my_list)
>>> my_2darray = np.array([[1,2,3],[4,5,6]])
```

Selecting Numpy Array Elements
Index starts at 0

Subset
`>>> my_array[1]`	Select item at index 1
` 2`	

Slice
`>>> my_array[0:2]`	Select items at index 0 and 1
` array([1, 2])`	

Subset 2D Numpy arrays
`>>> my_2darray[:,0]`	`my_2darray[rows, columns]`
` array([1, 4])`	

Numpy Array Operations

```
>>> my_array > 3
 array([False, False, False,  True], dtype=bool)
>>> my_array * 2
 array([2, 4, 6, 8])
>>> my_array + np.array([5, 6, 7, 8])
 array([6, 8, 10, 12])
```

Numpy Array Functions

`>>> my_array.shape`	Get the dimensions of the array
`>>> np.append(other_array)`	Append items to an array
`>>> np.insert(my_array, 1, 5)`	Insert items in an array
`>>> np.delete(my_array,[1])`	Delete items in an array
`>>> np.mean(my_array)`	Mean of the array
`>>> np.median(my_array)`	Median of the array
`>>> my_array.corrcoef()`	Correlation coefficient
`>>> np.std(my_array)`	Standard deviation

Data Science Cheat Sheet
Python - Intermediate

KEY BASICS, PRINTING AND GETTING HELP

This cheat sheet assumes you are familiar with the content of our Python Basics Cheat Sheet

s - A Python string variable
i - A Python integer variable
f - A Python float variable

l - A Python list variable
d - A Python dictionary variable

LISTS

l.pop(3) - Returns the fourth item from **l** and deletes it from the list

l.remove(x) - Removes the first item in **l** that is equal to **x**

l.reverse() - Reverses the order of the items in **l**

l[1::2] - Returns every second item from **l**, commencing from the **1**st item

l[-5:] - Returns the last 5 items from **l** specific axis

STRINGS

s.lower() - Returns a lowercase version of **s**

s.title() - Returns **s** with the first letter of every word capitalized

"23".zfill(4) - Returns **"0023"** by left-filling the string with **0**'s to make it's length **4**.

s.splitlines() - Returns a list by splitting the string on any newline characters.

Python strings share some common methods with lists

s[:5] - Returns the first **5** characters of **s**

"fri" + "end" - Returns **"friend"**

"end" in s - Returns **True** if the substring **"end"** is found in **s**

RANGE

Range objects are useful for creating sequences of integers for looping.

range(5) - Returns a sequence from **0** to **4**

range(2000,2018) - Returns a sequence from **2000** to **2017**

range(0,11,2) - Returns a sequence from **0** to **10**, with each item incrementing by **2**

range(0,-10,-1) - Returns a sequence from **0** to **-9**

list(range(5)) - Returns a list from **0** to **4**

DICTIONARIES

max(d, key=d.get) - Return the key that corresponds to the largest value in **d**

min(d, key=d.get) - Return the key that corresponds to the smallest value in **d**

SETS

my_set = set(l) - Return a **set** object containing the unique values from **l**

len(my_set) - Returns the number of objects in **my_set** (or, the number of unique values from **l**)

a in my_set - Returns **True** if the value **a** exists in **my_set**

REGULAR EXPRESSIONS

import re - Import the Regular Expressions module

re.search("abc",s) - Returns a **match** object if the regex **"abc"** is found in **s**, otherwise **None**

re.sub("abc","xyz",s) - Returns a string where all instances matching regex **"abc"** are replaced by **"xyz"**

LIST COMPREHENSION

A one-line expression of a for loop

[i ** 2 for i in range(10)] - Returns a list of the squares of values from **0** to **9**

[s.lower() for s in l_strings] - Returns the list **l_strings**, with each item having had the **.lower()** method applied

[i for i in l_floats if i < 0.5] - Returns the items from **l_floats** that are less than **0.5**

FUNCTIONS FOR LOOPING

```
for i, value in enumerate(l):
    print("The value of item {} is {}".
    format(i,value))
```
- Iterate over the list **l**, printing the index location of each item and its value

```
for one, two in zip(l_one,l_two):
    print("one: {}, two: {}".format(one,two))
```
- Iterate over two lists, **l_one** and **l_two** and print each value

```
while x < 10:
    x += 1
```
- Run the code in the body of the loop until the value of **x** is no longer less than **10**

DATETIME

import datetime as dt - Import the **datetime** module

now = dt.datetime.now() - Assign **datetime** object representing the current time to **now**

wks4 = dt.datetime.timedelta(weeks=4)
- Assign a **timedelta** object representing a timespan of 4 weeks to **wks4**

now - wks4 - Return a **datetime** object representing the time 4 weeks prior to **now**

newyear_2020 = dt.datetime(year=2020, month=12, day=31) - Assign a **datetime** object representing December 25, 2020 to **newyear_2020**

newyear_2020.strftime("%A, %b %d, %Y")
- Returns **"Thursday, Dec 31, 2020"**

dt.datetime.strptime('Dec 31, 2020',"%b %d, %Y") - Return a **datetime** object representing December 31, 2020

RANDOM

import random - Import the **random** module

random.random() - Returns a random float between **0.0** and **1.0**

random.randint(0,10) - Returns a random integer between **0** and **10**

random.choice(l) - Returns a random item from the list **l**

COUNTER

from collections import Counter - Import the **Counter** class

c = Counter(l) - Assign a **Counter** (dict-like) object with the counts of each unique item from **l**, to **c**

c.most_common(3) - Return the 3 most common items from **l**

TRY/EXCEPT

Catch and deal with Errors

l_ints = [1, 2, 3, "", 5] - Assign a list of integers with one missing value to **l_ints**

```
l_floats = []
for i in l_ints:
    try:
        l_floats.append(float(i))
    except:
        l_floats.append(i)
```
- Convert each value of **l_ints** to a float, catching and handling **ValueError: could not convert string to float:** where values are missing.
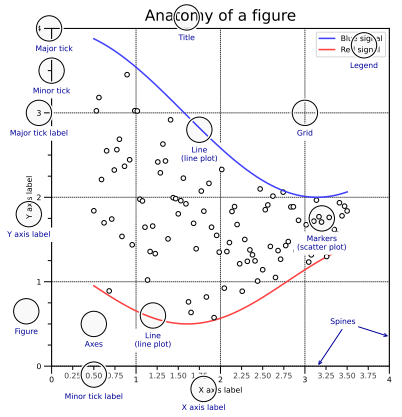
matplotlib

Cheat sheet

Version 3.5.0

Quick start `API`

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt

X = np.linspace(0, 2*np.pi, 100)
Y = np.cos(X)

fig, ax = plt.subplots()
ax.plot(X, Y, color='green')

fig.savefig("figure.pdf")
fig.show()
```

Anatomy of a figure

Anatomy of a figure

Subplots layout `API`

```
subplot[s](rows,cols,…)
fig, axs = plt.subplots(3, 3)
```

```
G = gridspec(rows,cols,…)
ax = G[0,:]
```

```
ax.inset_axes(extent)
```

```
d=make_axes_locatable(ax)
ax = d.new_horizontal('10%')
```

Getting help

- 🌐 matplotlib.org
- github.com/matplotlib/matplotlib/issues
- discourse.matplotlib.org
- stackoverflow.com/questions/tagged/matplotlib
- gitter.im/matplotlib
- twitter.com/matplotlib
- ✉ Matplotlib users mailing list

Basic plots `API`

plot([X],Y,[fmt],…) `API`
X, **Y**, fmt, color, marker, linestyle

scatter(X,Y,…) `API`
X, **Y**, [s]sizes, [c]colors, marker, cmap

bar[h](x,height,…) `API`
x, **height**, width, bottom, align, color

imshow(Z,…) `API`
Z, cmap, interpolation, extent, origin

contour[f]([X],[Y],Z,…) `API`
X, Y, **Z**, levels, colors, extent, origin

pcolormesh([X],[Y],Z,…) `API`
X, Y, **Z**, vmin, vmax, cmap

quiver([X],[Y],U,V,…) `API`
X, Y, **U**, **V**, C, units, angles

pie(X,…) `API`
Z, explode, labels, colors, radius

text(x,y,text,…) `API`
x, **y**, **text**, va, ha, size, weight, transform

fill[_between][x](…) `API`
X, Y1, Y2, color, where

Advanced plots

step(X,Y,[fmt],…) `API`
X, **Y**, fmt, color, marker, where

boxplot(X,…) `API`
X, notch, sym, bootstrap, widths

errorbar(X,Y,xerr,yerr,…) `API`
X, **Y**, xerr, yerr, fmt

hist(X, bins, …) `API`
X, bins, range, density, weights

violinplot(D,…) `API`
D, positions, widths, vert

barbs([X],[Y], U, V, …) `API`
X, Y, **U**, **V**, C, length, pivot, sizes

eventplot(positions,…) `API`
positions, orientation, lineoffsets

hexbin(X,Y,C,…) `API`
X, **Y**, C, gridsize, bins

Scales `API`

```
ax.set_[xy]scale(scale,…)
```

linear — any values
log — values > 0
symlog — any values
logit — 0 < values < 1

Projections `API`

```
subplot(…,projection=p)
```
p='polar'
p='3d'

p=Orthographic()
from cartopy.crs import Cartographic

Lines `API`

linestyle or ls

"-" ":" "--" "-." (0,(0.01,2))

capstyle or dash_capstyle

"butt" "round" "projecting"

Markers `API`

```
'.' ',' 'o' 's' 'P' 'X' '*' 'p' 'D' '<' '>' '^' 'v'
'1' '2' '3' '4' '+' 'x' '|' '_' 4 5 6 7
'$♠$''$♣$''$♥$''$♦$'→'←'↑'↓'$○$''$○$''$○$''$○$'
```

markevery

10
[0, -1]
(25, 5)
[0, 25, -1]

Colors `API`

```
C0 C1 C2 C3 C4 C5 C6 C7 C8 C9    'Cn'
b g r c m y k w               'x'
DarkRed Firebrick Crimson IndianRed Salmon  'name'
(1,0,0) (1,0,0,0.75) (1,0,0.5) (1,0,0.25) (1,0,0)  (R,G,B[,A])
#FF0000 #FF0000BB #FF0000 #FF000088 #FF000044  '#RRGGBB[AA]'
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0  'x.y'
```

Colormaps `API`

```
plt.get_cmap(name)
```

Uniform
viridis
magma
plasma

Sequential
Greys
YlOrBr
Wistia

Diverging
Spectral
coolwarm
RdGy

Qualitative
tab10
tab20

Cyclic
twilight

Tick locators `API`

```
from matplotlib import ticker
ax.[xy]axis.set_[minor|major]_locator(locator)
```

```
ticker.NullLocator()
```

```
ticker.MultipleLocator(0.5)
```

```
ticker.FixedLocator([0, 1, 5])
```

```
ticker.LinearLocator(numticks=3)
```

```
ticker.IndexLocator(base=0.5, offset=0.25)
```

```
ticker.AutoLocator()
```

```
ticker.MaxNLocator(n=4)
```

```
ticker.LogLocator(base=10, numticks=15)
```

Tick formatters `API`

```
from matplotlib import ticker
ax.[xy]axis.set_[minor|major]_formatter(formatter)
```

```
ticker.NullFormatter()
```

```
ticker.FixedFormatter(['zero', 'one', 'two', …])
```

```
ticker.FuncFormatter(lambda x, pos: "[%.2f]" % x)
```

```
ticker.FormatStrFormatter('>%d<')
```

```
ticker.ScalarFormatter()
```

```
ticker.StrMethodFormatter('{x}')
```

```
ticker.PercentFormatter(xmax=5)
```

Ornaments

```
ax.legend(…)   API
```
handles, labels, loc, title, frameon
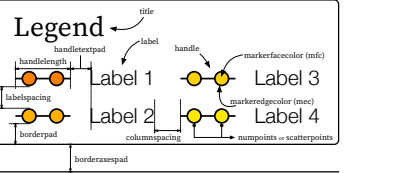
```
ax.colorbar(…)   API
```
mappable, ax, cax, orientation

```
ax.annotate(…)   API
```
text, xy, xytext, xycoords, textcoords, arrowprops
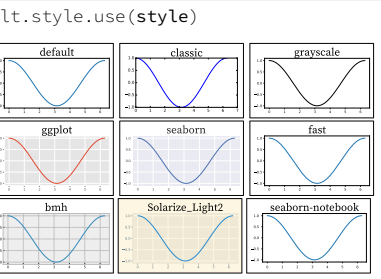
text
xytext
textcoords
xy
xycoords

Event handling `API`

```
fig, ax = plt.subplots()
def on_click(event):
    print(event)
fig.canvas.mpl_connect(
    'button_press_event', on_click)
```

Animation `API`

```
import matplotlib.animation as mpla

T = np.linspace(0, 2*np.pi, 100)
S = np.sin(T)
line, = plt.plot(T, S)
def animate(i):
    line.set_ydata(np.sin(T+i/50))
anim = mpla.FuncAnimation(
    plt.gcf(), animate, interval=5)
plt.show()
```

Styles `API`

```
plt.style.use(style)
```

default | classic | grayscale
ggplot | seaborn | fast
bmh | Solarize_Light2 | seaborn-notebook

Quick reminder

```
ax.grid()
ax.set_[xy]lim(vmin, vmax)
ax.set_[xy]label(label)
ax.set_[xy]ticks(ticks, [labels])
ax.set_[xy]ticklabels(labels)
ax.set_title(title)
ax.tick_params(width=10, …)
ax.set_axis_[on|off]()

fig.suptitle(title)
fig.tight_layout()
plt.gcf(), plt.gca()
mpl.rc('axes', linewidth=1, …)
[fig|ax].patch.set_alpha(0)
text=r'$\frac{-e^{i\pi}}{2^n}$'
```

Keyboard shortcuts `API`

ctrl + s	Save	ctrl + w	Close plot
r	Reset view	f	Fullscreen 0/1
f	View forward	b	View back
p	Pan view	o	Zoom to rect
x	X pan/zoom	y	Y pan/zoom
g	Minor grid 0/1	G	Major grid 0/1
l	X axis log/linear	L	Y axis log/linear

Ten simple rules `READ`

1. Know Your Audience
2. Identify Your Message
3. Adapt the Figure
4. Captions Are Not Optional
5. Do Not Trust the Defaults
6. Use Color Effectively
7. Do Not Mislead the Reader
8. Avoid "Chartjunk"
9. Message Trumps Beauty
10. Get the Right Tool

Axes adjustments `API`

plt.subplots_adjust(...)

Extent & origin `API`

ax.imshow(extent=..., origin=...)

Text alignments `API`

ax.text(..., ha=... , va=..., ...)

Text parameters `API`

ax.text(..., family=..., size=..., weight=...)
ax.text(..., fontproperties=...)

The quick brown fox	xx-large (1.73)
The quick brown fox	x-large (1.44)
The quick brown fox	large (1.20)
The quick brown fox	medium (1.00)
The quick brown fox	small (0.83)
The quick brown fox	x-small (0.69)
The quick brown fox	xx-small (0.58)
The quick brown fox jumps over the lazy dog	black (900)
The quick brown fox jumps over the lazy dog	bold (700)
The quick brown fox jumps over the lazy dog	semibold (600)
The quick brown fox jumps over the lazy dog	normal (400)
The quick brown fox jumps over the lazy dog	ultralight (100)
The quick brown fox jumps over the lazy dog	monospace
The quick brown fox jumps over the lazy dog	serif
The quick brown fox jumps over the lazy dog	sans
The quick brown fox jumps over the lazy dog	cursive
The quick brown fox jumps over the lazy dog	italic normal
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG	small-caps
The quick brown fox jumps over the lazy dog	normal

Uniform colormaps

viridis
plasma
inferno
magma
cividis

Sequential colormaps

Greys, Purples, Blues, Greens, Oranges, Reds, YlOrBr, YlOrRd, OrRd, PuRd, RdPu, BuPu, GnBu, PuBu, YlGnBu, PuBuGn, BuGn, YlGn

Diverging colormaps

PiYG, PRGn, BrBG, PuOr, RdGy, RdBu, RdYlBu, RdYlGn, Spectral, coolwarm, bwr, seismic

Qualitative colormaps

Pastel1, Pastel2, Paired, Accent, Dark2, Set1, Set2, Set3, tab10, tab20, tab20b, tab20c

Miscellaneous colormaps

terrain, ocean, cubehelix, rainbow, twilight

Color names `API`

Image interpolation `API`

None, none, nearest, bilinear, bicubic, spline16, spline36, hanning, hamming, hermite, kaiser, quadric, catrom, gaussian, bessel, mitchell, sinc, lanczos

Legend placement

ax.legend(loc="string", bbox_to_anchor=(x,y))

2: upper left	9: upper center	1: upper right
6: center left	10: center	7: center right
3: lower left	8: lower center	4: lower right

A: upper right / (-0.1,0.9)
B: center right / (-0.1,0.5)
C: lower right / (-0.1,0.1)
D: upper left / (0.1,-0.1)
E: upper center / (0.5,-0.1)
F: upper right / (0.9,-0.1)
G: lower left / (1.1,0.1)
H: center left / (1.1,0.5)
I: upper left / (1.1,0.9)
J: lower right / (0.9,1.1)
K: lower center / (0.5,1.1)
L: lower left / (0.1,1.1)

Annotation connection styles `API`

Annotation arrow styles `API`

How do I …

… resize a figure?
→ fig.set_size_inches(w, h)
… save a figure?
→ fig.savefig("figure.pdf")
… save a transparent figure?
→ fig.savefig("figure.pdf", transparent=True)
… clear a figure/an axes?
→ fig.clear() / ax.clear()
… close all figures?
→ plt.close("all")
… remove ticks?
→ ax.set_[xy]ticks([])
… remove tick labels ?
→ ax.set_[xy]ticklabels([])
… rotate tick labels ?
→ ax.tick_params(axis="x", rotation=90)
… hide top spine?
→ ax.spines['top'].set_visible(False)
… hide legend border?
→ ax.legend(frameon=False)
… show error as shaded region?
→ ax.fill_between(X, Y+error, Y-error)
… draw a rectangle?
→ ax.add_patch(plt.Rectangle((0, 0), 1, 1)
… draw a vertical line?
→ ax.axvline(x=0.5)
… draw outside frame?
→ ax.plot(..., clip_on=False)
… use transparency?
→ ax.plot(..., alpha=0.25)
… convert an RGB image into a gray image?
→ gray = 0.2989*R + 0.5870*G + 0.1140*B
… set figure background color?
→ fig.patch.set_facecolor("grey")
… get a reversed colormap?
→ plt.get_cmap("viridis_r")
… get a discrete colormap?
→ plt.get_cmap("viridis", 10)
… show a figure for one second?
→ fig.show(block=False), time.sleep(1)

Performance tips

scatter(X, Y)	slow
plot(X, Y, marker="o", ls="")	fast
for i in range(n): plot(X[i])	slow
plot(sum([x+[None] for x in X],[]))	fast
cla(), imshow(…), canvas.draw()	slow
im.set_data(…), canvas.draw()	fast

Beyond Matplotlib

Seaborn: Statistical Data Visualization
Cartopy: Geospatial Data Processing
yt: Volumetric data Visualization
mpld3: Bringing Matplotlib to the browser
Datashader: Large data processing pipeline
plotnine: A Grammar of Graphics for Python

Beginner's Python Cheat Sheet

Variables and Strings

Variables are used to store values. A string is a series of characters, surrounded by single or double quotes.

Hello world

```python
print("Hello world!")
```

Hello world with a variable

```python
msg = "Hello world!"
print(msg)
```

Concatenation (combining strings)

```python
first_name = 'albert'
last_name = 'einstein'
full_name = first_name + ' ' + last_name
print(full_name)
```

Lists

A list stores a series of items in a particular order. You access items using an index, or within a loop.

Make a list

```python
bikes = ['trek', 'redline', 'giant']
```

Get the first item in a list

```python
first_bike = bikes[0]
```

Get the last item in a list

```python
last_bike = bikes[-1]
```

Looping through a list

```python
for bike in bikes:
    print(bike)
```

Adding items to a list

```python
bikes = []
bikes.append('trek')
bikes.append('redline')
bikes.append('giant')
```

Making numerical lists

```python
squares = []
for x in range(1, 11):
    squares.append(x**2)
```

Lists (cont.)

List comprehensions

```python
squares = [x**2 for x in range(1, 11)]
```

Slicing a list

```python
finishers = ['sam', 'bob', 'ada', 'bea']
first_two = finishers[:2]
```

Copying a list

```python
copy_of_bikes = bikes[:]
```

Tuples

Tuples are similar to lists, but the items in a tuple can't be modified.

Making a tuple

```python
dimensions = (1920, 1080)
```

If statements

If statements are used to test for particular conditions and respond appropriately.

Conditional tests

```
equals            x == 42
not equal         x != 42
greater than      x > 42
  or equal to     x >= 42
less than         x < 42
  or equal to     x <= 42
```

Conditional test with lists

```python
'trek' in bikes
'surly' not in bikes
```

Assigning boolean values

```python
game_active = True
can_edit = False
```

A simple if test

```python
if age >= 18:
    print("You can vote!")
```

If-elif-else statements

```python
if age < 4:
    ticket_price = 0
elif age < 18:
    ticket_price = 10
else:
    ticket_price = 15
```

Dictionaries

Dictionaries store connections between pieces of information. Each item in a dictionary is a key-value pair.

A simple dictionary

```python
alien = {'color': 'green', 'points': 5}
```

Accessing a value

```python
print("The alien's color is " + alien['color'])
```

Adding a new key-value pair

```python
alien['x_position'] = 0
```

Looping through all key-value pairs

```python
fav_numbers = {'eric': 17, 'ever': 4}
for name, number in fav_numbers.items():
    print(name + ' loves ' + str(number))
```

Looping through all keys

```python
fav_numbers = {'eric': 17, 'ever': 4}
for name in fav_numbers.keys():
    print(name + ' loves a number')
```

Looping through all the values

```python
fav_numbers = {'eric': 17, 'ever': 4}
for number in fav_numbers.values():
    print(str(number) + ' is a favorite')
```

User input

Your programs can prompt the user for input. All input is stored as a string.

Prompting for a value

```python
name = input("What's your name? ")
print("Hello, " + name + "!")
```

Prompting for numerical input

```python
age = input("How old are you? ")
age = int(age)

pi = input("What's the value of pi? ")
pi = float(pi)
```

Python Crash Course

Covers Python 3 and Python 2
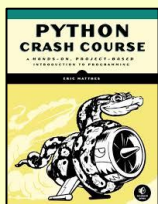
While loops

A while loop repeats a block of code as long as a certain condition is true.

A simple while loop

```python
current_value = 1
while current_value <= 5:
    print(current_value)
    current_value += 1
```

Letting the user choose when to quit

```python
msg = ''
while msg != 'quit':
    msg = input("What's your message? ")
    print(msg)
```

Functions

Functions are named blocks of code, designed to do one specific job. Information passed to a function is called an argument, and information received by a function is called a parameter.

A simple function

```python
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

Passing an argument

```python
def greet_user(username):
    """Display a personalized greeting."""
    print("Hello, " + username + "!")

greet_user('jesse')
```

Default values for parameters

```python
def make_pizza(topping='bacon'):
    """Make a single-topping pizza."""
    print("Have a " + topping + " pizza!")

make_pizza()
make_pizza('pepperoni')
```

Returning a value

```python
def add_numbers(x, y):
    """Add two numbers and return the sum."""
    return x + y

sum = add_numbers(3, 5)
print(sum)
```

Classes

A class defines the behavior of an object and the kind of information an object can store. The information in a class is stored in attributes, and functions that belong to a class are called methods. A child class inherits the attributes and methods from its parent class.

Creating a dog class

```python
class Dog():
    """Represent a dog."""

    def __init__(self, name):
        """Initialize dog object."""
        self.name = name

    def sit(self):
        """Simulate sitting."""
        print(self.name + " is sitting.")

my_dog = Dog('Peso')

print(my_dog.name + " is a great dog!")
my_dog.sit()
```

Inheritance

```python
class SARDog(Dog):
    """Represent a search dog."""

    def __init__(self, name):
        """Initialize the sardog."""
        super().__init__(name)

    def search(self):
        """Simulate searching."""
        print(self.name + " is searching.")

my_dog = SARDog('Willie')

print(my_dog.name + " is a search dog.")
my_dog.sit()
my_dog.search()
```

Infinite Skills

If you had infinite programming skills, what would you build?

As you're learning to program, it's helpful to think about the real-world projects you'd like to create. It's a good habit to keep an "ideas" notebook that you can refer to whenever you want to start a new project. If you haven't done so already, take a few minutes and describe three projects you'd like to create.

Working with files

Your programs can read from files and write to files. Files are opened in read mode ('r') by default, but can also be opened in write mode ('w') and append mode ('a').

Reading a file and storing its lines

```python
filename = 'siddhartha.txt'
with open(filename) as file_object:
    lines = file_object.readlines()

for line in lines:
    print(line)
```

Writing to a file

```python
filename = 'journal.txt'
with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
```

Appending to a file

```python
filename = 'journal.txt'
with open(filename, 'a') as file_object:
    file_object.write("\nI love making games.")
```

Exceptions

Exceptions help you respond appropriately to errors that are likely to occur. You place code that might cause an error in the try block. Code that should run in response to an error goes in the except block. Code that should run only if the try block was successful goes in the else block.

Catching an exception

```python
prompt = "How many tickets do you need? "
num_tickets = input(prompt)

try:
    num_tickets = int(num_tickets)
except ValueError:
    print("Please try again.")
else:
    print("Your tickets are printing.")
```

Zen of Python

Simple is better than complex

If you have a choice between a simple and a complex solution, and both work, use the simple solution. Your code will be easier to maintain, and it will be easier for you and others to build on that code later on.

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet - Lists

What are lists?

A list stores a series of items in a particular order. Lists allow you to store sets of information in one place, whether you have just a few items or millions of items. Lists are one of Python's most powerful features readily accessible to new programmers, and they tie together many important concepts in programming.

Defining a list

Use square brackets to define a list, and use commas to separate individual items in the list. Use plural names for lists, to make your code easier to read.

Making a list

```
users = ['val', 'bob', 'mia', 'ron', 'ned']
```

Accessing elements

Individual elements in a list are accessed according to their position, called the index. The index of the first element is 0, the index of the second element is 1, and so forth. Negative indices refer to items at the end of the list. To get a particular element, write the name of the list and then the index of the element in square brackets.

Getting the first element

```
first_user = users[0]
```

Getting the second element

```
second_user = users[1]
```

Getting the last element

```
newest_user = users[-1]
```

Modifying individual items

Once you've defined a list, you can change individual elements in the list. You do this by referring to the index of the item you want to modify.

Changing an element

```
users[0] = 'valerie'
users[-2] = 'ronald'
```

Adding elements

You can add elements to the end of a list, or you can insert them wherever you like in a list.

Adding an element to the end of the list

```
users.append('amy')
```

Starting with an empty list

```
users = []
users.append('val')
users.append('bob')
users.append('mia')
```

Inserting elements at a particular position

```
users.insert(0, 'joe')
users.insert(3, 'bea')
```

Removing elements

You can remove elements by their position in a list, or by the value of the item. If you remove an item by its value, Python removes only the first item that has that value.

Deleting an element by its position

```
del users[-1]
```

Removing an item by its value

```
users.remove('mia')
```

Popping elements

If you want to work with an element that you're removing from the list, you can "pop" the element. If you think of the list as a stack of items, pop() takes an item off the top of the stack. By default pop() returns the last element in the list, but you can also pop elements from any position in the list.

Pop the last item from a list

```
most_recent_user = users.pop()
print(most_recent_user)
```

Pop the first item in a list

```
first_user = users.pop(0)
print(first_user)
```

List length

The len() function returns the number of items in a list.

Find the length of a list

```
num_users = len(users)
print("We have " + str(num_users) + " users.")
```

Sorting a list

The sort() method changes the order of a list permanently. The sorted() function returns a copy of the list, leaving the original list unchanged. You can sort the items in a list in alphabetical order, or reverse alphabetical order. You can also reverse the original order of the list. Keep in mind that lowercase and uppercase letters may affect the sort order.

Sorting a list permanently

```
users.sort()
```

Sorting a list permanently in reverse alphabetical order

```
users.sort(reverse=True)
```

Sorting a list temporarily

```
print(sorted(users))
print(sorted(users, reverse=True))
```

Reversing the order of a list

```
users.reverse()
```

Looping through a list

Lists can contain millions of items, so Python provides an efficient way to loop through all the items in a list. When you set up a loop, Python pulls each item from the list one at a time and stores it in a temporary variable, which you provide a name for. This name should be the singular version of the list name.

The indented block of code makes up the body of the loop, where you can work with each individual item. Any lines that are not indented run after the loop is completed.

Printing all items in a list

```
for user in users:
    print(user)
```

Printing a message for each item, and a separate message afterwards

```
for user in users:
    print("Welcome, " + user + "!")

print("Welcome, we're glad to see you all!")
```

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse
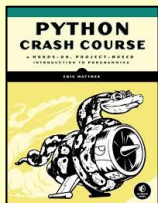
The range() function

You can use the range() function to work with a set of numbers efficiently. The range() function starts at 0 by default, and stops one number below the number passed to it. You can use the list() function to efficiently generate a large list of numbers.

Printing the numbers 0 to 1000

```python
for number in range(1001):
    print(number)
```

Printing the numbers 1 to 1000

```python
for number in range(1, 1001):
    print(number)
```

Making a list of numbers from 1 to a million

```python
numbers = list(range(1, 1000001))
```

Simple statistics

There are a number of simple statistics you can run on a list containing numerical data.

Finding the minimum value in a list

```python
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
youngest = min(ages)
```

Finding the maximum value

```python
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
oldest = max(ages)
```

Finding the sum of all values

```python
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
total_years = sum(ages)
```

Slicing a list

You can work with any set of elements from a list. A portion of a list is called a slice. To slice a list start with the index of the first item you want, then add a colon and the index after the last item you want. Leave off the first index to start at the beginning of the list, and leave off the last index to slice through the end of the list.

Getting the first three items

```python
finishers = ['kai', 'abe', 'ada', 'gus', 'zoe']
first_three = finishers[:3]
```

Getting the middle three items

```python
middle_three = finishers[1:4]
```

Getting the last three items

```python
last_three = finishers[-3:]
```

Copying a list

To copy a list make a slice that starts at the first item and ends at the last item. If you try to copy a list without using this approach, whatever you do to the copied list will affect the original list as well.

Making a copy of a list

```python
finishers = ['kai', 'abe', 'ada', 'gus', 'zoe']
copy_of_finishers = finishers[:]
```

List comprehensions

You can use a loop to generate a list based on a range of numbers or on another list. This is a common operation, so Python offers a more efficient way to do it. List comprehensions may look complicated at first; if so, use the for loop approach until you're ready to start using comprehensions.

To write a comprehension, define an expression for the values you want to store in the list. Then write a for loop to generate input values needed to make the list.

Using a loop to generate a list of square numbers

```python
squares = []
for x in range(1, 11):
    square = x**2
    squares.append(square)
```

Using a comprehension to generate a list of square numbers

```python
squares = [x**2 for x in range(1, 11)]
```

Using a loop to convert a list of names to upper case

```python
names = ['kai', 'abe', 'ada', 'gus', 'zoe']

upper_names = []
for name in names:
    upper_names.append(name.upper())
```

Using a comprehension to convert a list of names to upper case

```python
names = ['kai', 'abe', 'ada', 'gus', 'zoe']

upper_names = [name.upper() for name in names]
```

Styling your code
Readability counts

- Use four spaces per indentation level.
- Keep your lines to 79 characters or fewer.
- Use single blank lines to group parts of your program visually.

Tuples

A tuple is like a list, except you can't change the values in a tuple once it's defined. Tuples are good for storing information that shouldn't be changed throughout the life of a program. Tuples are designated by parentheses instead of square brackets. (You can overwrite an entire tuple, but you can't change the individual elements in a tuple.)

Defining a tuple

```python
dimensions = (800, 600)
```

Looping through a tuple

```python
for dimension in dimensions:
    print(dimension)
```

Overwriting a tuple

```python
dimensions = (800, 600)
print(dimensions)

dimensions = (1200, 900)
```

Visualizing your code

When you're first learning about data structures such as lists, it helps to visualize how Python is working with the information in your program. pythontutor.com is a great tool for seeing how Python keeps track of the information in a list. Try running the following code on pythontutor.com, and then run your own code.

Build a list and print the items in the list

```python
dogs = []
dogs.append('willie')
dogs.append('hootz')
dogs.append('peso')
dogs.append('goblin')

for dog in dogs:
    print("Hello " + dog + "!")
print("I love these dogs!")

print("\nThese were my first two dogs:")
old_dogs = dogs[:2]
for old_dog in old_dogs:
    print(old_dog)

del dogs[0]
dogs.remove('peso')
print(dogs)
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Dictionaries

What are dictionaries?

Python's dictionaries allow you to connect pieces of related information. Each piece of information in a dictionary is stored as a key-value pair. When you provide a key, Python returns the value associated with that key. You can loop through all the key-value pairs, all the keys, or all the values.

Defining a dictionary

Use curly braces to define a dictionary. Use colons to connect keys and values, and use commas to separate individual key-value pairs.

Making a dictionary

```python
alien_0 = {'color': 'green', 'points': 5}
```

Accessing values

To access the value associated with an individual key give the name of the dictionary and then place the key in a set of square brackets. If the key you're asking for is not in the dictionary, an error will occur.

You can also use the get() method, which returns None instead of an error if the key doesn't exist. You can also specify a default value to use if the key is not in the dictionary.

Getting the value associated with a key

```python
alien_0 = {'color': 'green', 'points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

Getting the value with get()

```python
alien_0 = {'color': 'green'}

alien_color = alien_0.get('color')
alien_points = alien_0.get('points', 0)

print(alien_color)
print(alien_points)
```

Adding new key-value pairs

You can store as many key-value pairs as you want in a dictionary, until your computer runs out of memory. To add a new key-value pair to an existing dictionary give the name of the dictionary and the new key in square brackets, and set it equal to the new value.

This also allows you to start with an empty dictionary and add key-value pairs as they become relevant.

Adding a key-value pair

```python
alien_0 = {'color': 'green', 'points': 5}

alien_0['x'] = 0
alien_0['y'] = 25
alien_0['speed'] = 1.5
```

Adding to an empty dictionary

```python
alien_0 = {}
alien_0['color'] = 'green'
alien_0['points'] = 5
```

Modifying values

You can modify the value associated with any key in a dictionary. To do so give the name of the dictionary and enclose the key in square brackets, then provide the new value for that key.

Modifying values in a dictionary

```python
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

# Change the alien's color and point value.
alien_0['color'] = 'yellow'
alien_0['points'] = 10
print(alien_0)
```

Removing key-value pairs

You can remove any key-value pair you want from a dictionary. To do so use the del keyword and the dictionary name, followed by the key in square brackets. This will delete the key and its associated value.

Deleting a key-value pair

```python
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

del alien_0['points']
print(alien_0)
```

Visualizing dictionaries

Try running some of these examples on pythontutor.com.

Looping through a dictionary

You can loop through a dictionary in three ways: you can loop through all the key-value pairs, all the keys, or all the values.

A dictionary only tracks the connections between keys and values; it doesn't track the order of items in the dictionary. If you want to process the information in order, you can sort the keys in your loop.

Looping through all key-value pairs

```python
# Store people's favorite languages.
fav_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

# Show each person's favorite language.
for name, language in fav_languages.items():
    print(name + ": " + language)
```

Looping through all the keys

```python
# Show everyone who's taken the survey.
for name in fav_languages.keys():
    print(name)
```

Looping through all the values

```python
# Show all the languages that have been chosen.
for language in fav_languages.values():
    print(language)
```

Looping through all the keys in order

```python
# Show each person's favorite language,
#   in order by the person's name.
for name in sorted(fav_languages.keys()):
    print(name + ": " + language)
```

Dictionary length

You can find the number of key-value pairs in a dictionary.

Finding a dictionary's length

```python
num_responses = len(fav_languages)
```

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse
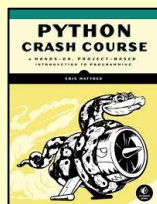
Nesting — A list of dictionaries

It's sometimes useful to store a set of dictionaries in a list; this is called nesting.

Storing dictionaries in a list

```python
# Start with an empty list.
users = []

# Make a new user, and add them to the list.
new_user = {
    'last': 'fermi',
    'first': 'enrico',
    'username': 'efermi',
    }
users.append(new_user)

# Make another new user, and add them as well.
new_user = {
    'last': 'curie',
    'first': 'marie',
    'username': 'mcurie',
    }
users.append(new_user)

# Show all information about each user.
for user_dict in users:
    for k, v in user_dict.items():
        print(k + ": " + v)
    print("\n")
```

You can also define a list of dictionaries directly, without using append():

```python
# Define a list of users, where each user
#   is represented by a dictionary.
users = [
    {
        'last': 'fermi',
        'first': 'enrico',
        'username': 'efermi',
    },
    {
        'last': 'curie',
        'first': 'marie',
        'username': 'mcurie',
    },
]

# Show all information about each user.
for user_dict in users:
    for k, v in user_dict.items():
        print(k + ": " + v)
    print("\n")
```

Nesting — Lists in a dictionary

Storing a list inside a dictionary allows you to associate more than one value with each key.

Storing lists in a dictionary

```python
# Store multiple languages for each person.
fav_languages = {
    'jen': ['python', 'ruby'],
    'sarah': ['c'],
    'edward': ['ruby', 'go'],
    'phil': ['python', 'haskell'],
}

# Show all responses for each person.
for name, langs in fav_languages.items():
    print(name + ": ")
    for lang in langs:
        print("- " + lang)
```

Nesting — A dictionary of dictionaries

You can store a dictionary inside another dictionary. In this case each value associated with a key is itself a dictionary.

Storing dictionaries in a dictionary

```python
users = {
    'aeinstein': {
        'first': 'albert',
        'last': 'einstein',
        'location': 'princeton',
        },
    'mcurie': {
        'first': 'marie',
        'last': 'curie',
        'location': 'paris',
        },
}

for username, user_dict in users.items():
    print("\nUsername: " + username)
    full_name = user_dict['first'] + " "
    full_name += user_dict['last']
    location = user_dict['location']

    print("\tFull name: " + full_name.title())
    print("\tLocation: " + location.title())
```

Levels of nesting

Nesting is extremely useful in certain situations. However, be aware of making your code overly complex. If you're nesting items much deeper than what you see here there are probably simpler ways of managing your data, such as using classes.

Using an OrderedDict

Standard Python dictionaries don't keep track of the order in which keys and values are added; they only preserve the association between each key and its value. If you want to preserve the order in which keys and values are added, use an OrderedDict.

Preserving the order of keys and values

```python
from collections import OrderedDict

# Store each person's languages, keeping
#  track of who respoded first.
fav_languages = OrderedDict()

fav_languages['jen'] = ['python', 'ruby']
fav_languages['sarah'] = ['c']
fav_languages['edward'] = ['ruby', 'go']
fav_languages['phil'] = ['python', 'haskell']

# Display the results, in the same order they
#  were entered.
for name, langs in fav_languages.items():
    print(name + ":")
    for lang in langs:
        print("- " + lang)
```

Generating a million dictionaries

You can use a loop to generate a large number of dictionaries efficiently, if all the dictionaries start out with similar data.

A million aliens

```python
aliens = []

# Make a million green aliens, worth 5 points
#  each. Have them all start in one row.
for alien_num in range(1000000):
    new_alien = {}
    new_alien['color'] = 'green'
    new_alien['points'] = 5
    new_alien['x'] = 20 * alien_num
    new_alien['y'] = 0
    aliens.append(new_alien)

# Prove the list contains a million aliens.
num_aliens = len(aliens)

print("Number of aliens created:")
print(num_aliens)
```

Beginner's Python Cheat Sheet — If Statements and While Loops

What are if statements? What are while loops?

If statements allow you to examine the current state of a program and respond appropriately to that state. You can write a simple if statement that checks one condition, or you can create a complex series of if statements that idenitfy the exact conditions you're looking for.

While loops run as long as certain conditions remain true. You can use while loops to let your programs run as long as your users want them to.

Conditional Tests

A conditional test is an expression that can be evaluated as True or False. Python uses the values True and False to decide whether the code in an if statement should be executed.

Checking for equality
A single equal sign assigns a value to a variable. A double equal sign (==) checks whether two values are equal.

```
>>> car = 'bmw'
>>> car == 'bmw'
True
>>> car = 'audi'
>>> car == 'bmw'
False
```

Ignoring case when making a comparison

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

Checking for inequality

```
>>> topping = 'mushrooms'
>>> topping != 'anchovies'
True
```

Numerical comparisons
Testing numerical values is similar to testing string values.

Testing equality and inequality

```
>>> age = 18
>>> age == 18
True
>>> age != 18
False
```

Comparison operators

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

Checking multiple conditions
You can check multiple conditions at the same time. The and operator returns True if all the conditions listed are True. The or operator returns True if any condition is True.

Using and to check multiple conditions

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 and age_1 >= 21
False
>>> age_1 = 23
>>> age_0 >= 21 and age_1 >= 21
True
```

Using or to check multiple conditions

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 or age_1 >= 21
True
>>> age_0 = 18
>>> age_0 >= 21 or age_1 >= 21
False
```

Boolean values
A boolean value is either True or False. Variables with boolean values are often used to keep track of certain conditions within a program.

Simple boolean values

```
game_active = True
can_edit = False
```

If statements
Several kinds of if statements exist. Your choice of which to use depends on the number of conditions you need to test. You can have as many elif blocks as you need, and the else block is always optional.

Simple if statement

```
age = 19

if age >= 18:
    print("You're old enough to vote!")
```

If-else statements

```
age = 17

if age >= 18:
    print("You're old enough to vote!")
else:
    print("You can't vote yet.")
```

The if-elif-else chain

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 5
else:
    price = 10

print("Your cost is $" + str(price) + ".")
```

Conditional tests with lists
You can easily test whether a certain value is in a list. You can also test whether a list is empty before trying to loop through the list.

Testing if a value is in a list

```
>>> players = ['al', 'bea', 'cyn', 'dale']
>>> 'al' in players
True
>>> 'eric' in players
False
```
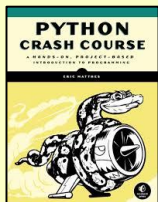
Conditional tests with lists (cont.)

Testing if a value is not in a list

```python
banned_users = ['ann', 'chad', 'dee']
user = 'erin'

if user not in banned_users:
    print("You can play!")
```

Checking if a list is empty

```python
players = []

if players:
    for player in players:
        print("Player: " + player.title())
else:
    print("We have no players yet!")
```

Accepting input

You can allow your users to enter input using the input() statement. In Python 3, all input is stored as a string.

Simple input

```python
name = input("What's your name? ")
print("Hello, " + name + ".")
```

Accepting numerical input

```python
age = input("How old are you? ")
age = int(age)

if age >= 18:
    print("\nYou can vote!")
else:
    print("\nYou can't vote yet.")
```

Accepting input in Python 2.7
Use raw_input() in Python 2.7. This function interprets all input as a string, just as input() does in Python 3.

```python
name = raw_input("What's your name? ")
print("Hello, " + name + ".")
```

While loops

A while loop repeats a block of code as long as a condition is True.

Counting to 5

```python
current_number = 1

while current_number <= 5:
    print(current_number)
    current_number += 1
```

While loops (cont.)

Letting the user choose when to quit

```python
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program. "

message = ""
while message != 'quit':
    message = input(prompt)

    if message != 'quit':
        print(message)
```

Using a flag

```python
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program. "

active = True
while active:
    message = input(prompt)

    if message == 'quit':
        active = False
    else:
        print(message)
```

Using break to exit a loop

```python
prompt = "\nWhat cities have you visited?"
prompt += "\nEnter 'quit' when you're done. "

while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        print("I've been to " + city + "!")
```

Accepting input with Sublime Text

Sublime Text doesn't run programs that prompt the user for input. You can use Sublime Text to write programs that prompt for input, but you'll need to run these programs from a terminal.

Breaking out of loops

You can use the break statement and the continue statement with any of Python's loops. For example you can use break to quit a for loop that's working through a list or a dictionary. You can use continue to skip over certain items when looping through a list or dictionary as well.

While loops (cont.)

Using continue in a loop

```python
banned_users = ['eve', 'fred', 'gary', 'helen']

prompt = "\nAdd a player to your team."
prompt += "\nEnter 'quit' when you're done. "

players = []
while True:
    player = input(prompt)
    if player == 'quit':
        break
    elif player in banned_users:
        print(player + " is banned!")
        continue
    else:
        players.append(player)

print("\nYour team:")
for player in players:
    print(player)
```

Avoiding infinite loops

Every while loop needs a way to stop running so it won't continue to run forever. If there's no way for the condition to become False, the loop will never stop running.

An infinite loop

```python
while True:
    name = input("\nWho are you? ")
    print("Nice to meet you, " + name + "!")
```

Removing all instances of a value from a list

The remove() method removes a specific value from a list, but it only removes the first instance of the value you provide. You can use a while loop to remove all instances of a particular value.

Removing all cats from a list of pets

```python
pets = ['dog', 'cat', 'dog', 'fish', 'cat',
        'rabbit', 'cat']
print(pets)

while 'cat' in pets:
    pets.remove('cat')

print(pets)
```

Beginner's Python Cheat Sheet — Functions

What are functions?

Functions are named blocks of code designed to do one specific job. Functions allow you to write code once that can then be run whenever you need to accomplish the same task. Functions can take in the information they need, and return the information they generate. Using functions effectively makes your programs easier to write, read, test, and fix.

Defining a function

The first line of a function is its definition, marked by the keyword def. The name of the function is followed by a set of parentheses and a colon. A docstring, in triple quotes, describes what the function does. The body of a function is indented one level.

To call a function, give the name of the function followed by a set of parentheses.

Making a function

```python
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

Passing information to a function

Information that's passed to a function is called an argument; information that's received by a function is called a parameter. Arguments are included in parentheses after the function's name, and parameters are listed in parentheses in the function's definition.

Passing a single argument

```python
def greet_user(username):
    """Display a simple greeting."""
    print("Hello, " + username + "!")

greet_user('jesse')
greet_user('diana')
greet_user('brandon')
```

Positional and keyword arguments

The two main kinds of arguments are positional and keyword arguments. When you use positional arguments Python matches the first argument in the function call with the first parameter in the function definition, and so forth.

With keyword arguments, you specify which parameter each argument should be assigned to in the function call. When you use keyword arguments, the order of the arguments doesn't matter.

Using positional arguments

```python
def describe_pet(animal, name):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')
```

Using keyword arguments

```python
def describe_pet(animal, name):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet(animal='hamster', name='harry')
describe_pet(name='willie', animal='dog')
```

Default values

You can provide a default value for a parameter. When function calls omit this argument the default value will be used. Parameters with default values must be listed after parameters without default values in the function's definition so positional arguments can still work correctly.

Using a default value

```python
def describe_pet(name, animal='dog'):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet('harry', 'hamster')
describe_pet('willie')
```

Using None to make an argument optional

```python
def describe_pet(animal, name=None):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    if name:
        print("Its name is " + name + ".")

describe_pet('hamster', 'harry')
describe_pet('snake')
```

Return values

A function can return a value or a set of values. When a function returns a value, the calling line must provide a variable in which to store the return value. A function stops running when it reaches a return statement.

Returning a single value

```python
def get_full_name(first, last):
    """Return a neatly formatted full name."""
    full_name = first + ' ' + last
    return full_name.title()

musician = get_full_name('jimi', 'hendrix')
print(musician)
```

Returning a dictionary

```python
def build_person(first, last):
    """Return a dictionary of information
    about a person.
    """
    person = {'first': first, 'last': last}
    return person

musician = build_person('jimi', 'hendrix')
print(musician)
```

Returning a dictionary with optional values

```python
def build_person(first, last, age=None):
    """Return a dictionary of information
    about a person.
    """
    person = {'first': first, 'last': last}
    if age:
        person['age'] = age
    return person

musician = build_person('jimi', 'hendrix', 27)
print(musician)

musician = build_person('janis', 'joplin')
print(musician)
```

Visualizing functions

Try running some of these examples on pythontutor.com.

Python Crash Course

Covers Python 3 and Python 2
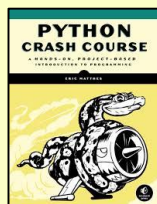
Passing a list to a function

You can pass a list as an argument to a function, and the function can work with the values in the list. Any changes the function makes to the list will affect the original list. You can prevent a function from modifying a list by passing a copy of the list as an argument.

Passing a list as an argument

```python
def greet_users(names):
    """Print a simple greeting to everyone."""
    for name in names:
        msg = "Hello, " + name + "!"
        print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

Allowing a function to modify a list
The following example sends a list of models to a function for printing. The original list is emptied, and the second list is filled.

```python
def print_models(unprinted, printed):
    """3d print a set of models."""
    while unprinted:
        current_model = unprinted.pop()
        print("Printing " + current_model)
        printed.append(current_model)

# Store some unprinted designs,
#  and print each of them.
unprinted = ['phone case', 'pendant', 'ring']
printed = []
print_models(unprinted, printed)

print("\nUnprinted:", unprinted)
print("Printed:", printed)
```

Preventing a function from modifying a list
The following example is the same as the previous one, except the original list is unchanged after calling print_models().

```python
def print_models(unprinted, printed):
    """3d print a set of models."""
    while unprinted:
        current_model = unprinted.pop()
        print("Printing " + current_model)
        printed.append(current_model)

# Store some unprinted designs,
#  and print each of them.
original = ['phone case', 'pendant', 'ring']
printed = []

print_models(original[:], printed)
print("\nOriginal:", original)
print("Printed:", printed)
```

Passing an arbitrary number of arguments

Sometimes you won't know how many arguments a function will need to accept. Python allows you to collect an arbitrary number of arguments into one parameter using the * operator. A parameter that accepts an arbitrary number of arguments must come last in the function definition.

The ** operator allows a parameter to collect an arbitrary number of keyword arguments.

Collecting an arbitrary number of arguments

```python
def make_pizza(size, *toppings):
    """Make a pizza."""
    print("\nMaking a " + size + " pizza.")
    print("Toppings:")
    for topping in toppings:
        print("- " + topping)

# Make three pizzas with different toppings.
make_pizza('small', 'pepperoni')
make_pizza('large', 'bacon bits', 'pineapple')
make_pizza('medium', 'mushrooms', 'peppers',
        'onions', 'extra cheese')
```

Collecting an arbitrary number of keyword arguments

```python
def build_profile(first, last, **user_info):
    """Build a user's profile dictionary."""
    # Build a dict with the required keys.
    profile = {'first': first, 'last': last}

    # Add any other keys and values.
    for key, value in user_info.items():
        profile[key] = value

    return profile

# Create two users with different kinds
#    of information.
user_0 = build_profile('albert', 'einstein',
        location='princeton')
user_1 = build_profile('marie', 'curie',
        location='paris', field='chemistry')

print(user_0)
print(user_1)
```

What's the best way to structure a function?

As you can see there are many ways to write and call a function. When you're starting out, aim for something that simply works. As you gain experience you'll develop an understanding of the more subtle advantages of different structures such as positional and keyword arguments, and the various approaches to importing functions. For now if your functions do what you need them to, you're doing well.

Modules

You can store your functions in a separate file called a module, and then import the functions you need into the file containing your main program. This allows for cleaner program files. (Make sure your module is stored in the same directory as your main program.)

Storing a function in a module
File: pizza.py

```python
def make_pizza(size, *toppings):
    """Make a pizza."""
    print("\nMaking a " + size + " pizza.")
    print("Toppings:")
    for topping in toppings:
        print("- " + topping)
```

Importing an entire module
File: making_pizzas.py
Every function in the module is available in the program file.

```python
import pizza

pizza.make_pizza('medium', 'pepperoni')
pizza.make_pizza('small', 'bacon', 'pineapple')
```

Importing a specific function
Only the imported functions are available in the program file.

```python
from pizza import make_pizza

make_pizza('medium', 'pepperoni')
make_pizza('small', 'bacon', 'pineapple')
```

Giving a module an alias

```python
import pizza as p

p.make_pizza('medium', 'pepperoni')
p.make_pizza('small', 'bacon', 'pineapple')
```

Giving a function an alias

```python
from pizza import make_pizza as mp

mp('medium', 'pepperoni')
mp('small', 'bacon', 'pineapple')
```

Importing all functions from a module
Don't do this, but recognize it when you see it in others' code. It can result in naming conflicts, which can cause errors.

```python
from pizza import *

make_pizza('medium', 'pepperoni')
make_pizza('small', 'bacon', 'pineapple')
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet - Classes

What are classes?

Classes are the foundation of object-oriented programming. Classes represent real-world things you want to model in your programs: for example dogs, cars, and robots. You use a class to make objects, which are specific instances of dogs, cars, and robots. A class defines the general behavior that a whole category of objects can have, and the information that can be associated with those objects.

Classes can inherit from each other – you can write a class that extends the functionality of an existing class. This allows you to code efficiently for a wide variety of situations.

Creating and using a class

Consider how we might model a car. What information would we associate with a car, and what behavior would it have? The information is stored in variables called attributes, and the behavior is represented by functions. Functions that are part of a class are called methods.

The Car class

```python
class Car():
    """A simple attempt to model a car."""

    def __init__(self, make, model, year):
        """Initialize car attributes."""
        self.make = make
        self.model = model
        self.year = year

        # Fuel capacity and level in gallons.
        self.fuel_capacity = 15
        self.fuel_level = 0

    def fill_tank(self):
        """Fill gas tank to capacity."""
        self.fuel_level = self.fuel_capacity
        print("Fuel tank is full.")

    def drive(self):
        """Simulate driving."""
        print("The car is moving.")
```

Creating and using a class (cont.)

Creating an object from a class

```python
my_car = Car('audi', 'a4', 2016)
```

Accessing attribute values

```python
print(my_car.make)
print(my_car.model)
print(my_car.year)
```

Calling methods

```python
my_car.fill_tank()
my_car.drive()
```

Creating multiple objects

```python
my_car = Car('audi', 'a4', 2016)
my_old_car = Car('subaru', 'outback', 2013)
my_truck = Car('toyota', 'tacoma', 2010)
```

Modifying attributes

You can modify an attribute's value directly, or you can write methods that manage updating values more carefully.

Modifying an attribute directly

```python
my_new_car = Car('audi', 'a4', 2016)
my_new_car.fuel_level = 5
```

Writing a method to update an attribute's value

```python
def update_fuel_level(self, new_level):
    """Update the fuel level."""
    if new_level <= self.fuel_capacity:
        self.fuel_level = new_level
    else:
        print("The tank can't hold that much!")
```

Writing a method to increment an attribute's value

```python
def add_fuel(self, amount):
    """Add fuel to the tank."""
    if (self.fuel_level + amount
            <= self.fuel_capacity):
        self.fuel_level += amount
        print("Added fuel.")
    else:
        print("The tank won't hold that much.")
```

Naming conventions

In Python class names are written in CamelCase and object names are written in lowercase with underscores. Modules that contain classes should still be named in lowercase with underscores.

Class inheritance

If the class you're writing is a specialized version of another class, you can use inheritance. When one class inherits from another, it automatically takes on all the attributes and methods of the parent class. The child class is free to introduce new attributes and methods, and override attributes and methods of the parent class.

To inherit from another class include the name of the parent class in parentheses when defining the new class.

The __init__() method for a child class

```python
class ElectricCar(Car):
    """A simple model of an electric car."""

    def __init__(self, make, model, year):
        """Initialize an electric car."""
        super().__init__(make, model, year)

        # Attributes specific to electric cars.
        # Battery capacity in kWh.
        self.battery_size = 70
        # Charge level in %.
        self.charge_level = 0
```

Adding new methods to the child class

```python
class ElectricCar(Car):
    --snip--
    def charge(self):
        """Fully charge the vehicle."""
        self.charge_level = 100
        print("The vehicle is fully charged.")
```

Using child methods and parent methods

```python
my_ecar = ElectricCar('tesla', 'model s', 2016)

my_ecar.charge()
my_ecar.drive()
```

Finding your workflow

There are many ways to model real world objects and situations in code, and sometimes that variety can feel overwhelming. Pick an approach and try it – if your first attempt doesn't work, try a different approach.

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse
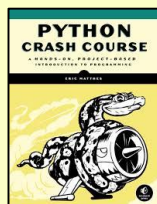
Class inheritance (cont.)

Overriding parent methods

```python
class ElectricCar(Car):
    --snip--
    def fill_tank(self):
        """Display an error message."""
        print("This car has no fuel tank!")
```

Instances as attributes

A class can have objects as attributes. This allows classes to work together to model complex situations.

A Battery class

```python
class Battery():
    """A battery for an electric car."""

    def __init__(self, size=70):
        """Initialize battery attributes."""
        # Capacity in kWh, charge level in %.
        self.size = size
        self.charge_level = 0

    def get_range(self):
        """Return the battery's range."""
        if self.size == 70:
            return 240
        elif self.size == 85:
            return 270
```

Using an instance as an attribute

```python
class ElectricCar(Car):
    --snip--

    def __init__(self, make, model, year):
        """Initialize an electric car."""
        super().__init__(make, model, year)

        # Attribute specific to electric cars.
        self.battery = Battery()

    def charge(self):
        """Fully charge the vehicle."""
        self.battery.charge_level = 100
        print("The vehicle is fully charged.")
```

Using the instance

```python
my_ecar = ElectricCar('tesla', 'model x', 2016)

my_ecar.charge()
print(my_ecar.battery.get_range())
my_ecar.drive()
```

Importing classes

Class files can get long as you add detailed information and functionality. To help keep your program files uncluttered, you can store your classes in modules and import the classes you need into your main program.

Storing classes in a file
car.py

```python
"""Represent gas and electric cars."""

class Car():
    """A simple attempt to model a car."""
    --snip--

class Battery():
    """A battery for an electric car."""
    --snip--

class ElectricCar(Car):
    """A simple model of an electric car."""
    --snip--
```

Importing individual classes from a module
my_cars.py

```python
from car import Car, ElectricCar

my_beetle = Car('volkswagen', 'beetle', 2016)
my_beetle.fill_tank()
my_beetle.drive()

my_tesla = ElectricCar('tesla', 'model s', 2016)
my_tesla.charge()
my_tesla.drive()
```

Importing an entire module

```python
import car

my_beetle = car.Car(
        'volkswagen', 'beetle', 2016)
my_beetle.fill_tank()
my_beetle.drive()

my_tesla = car.ElectricCar(
        'tesla', 'model s', 2016)
my_tesla.charge()
my_tesla.drive()
```

Importing all classes from a module
(Don't do this, but recognize it when you see it.)

```python
from car import *

my_beetle = Car('volkswagen', 'beetle', 2016)
```

Classes in Python 2.7

Classes should inherit from object

```python
class ClassName(object):
```

The Car class in Python 2.7

```python
class Car(object):
```

Child class __init__() method is different

```python
class ChildClassName(ParentClass):
    def __init__(self):
        super(ClassName, self).__init__()
```

The ElectricCar class in Python 2.7

```python
class ElectricCar(Car):
    def __init__(self, make, model, year):
        super(ElectricCar, self).__init__(
            make, model, year)
```

Storing objects in a list

A list can hold as many items as you want, so you can make a large number of objects from a class and store them in a list.
*   Here's an example showing how to make a fleet of rental cars, and make sure all the cars are ready to drive.*

A fleet of rental cars

```python
from car import Car, ElectricCar

# Make lists to hold a fleet of cars.
gas_fleet = []
electric_fleet = []

# Make 500 gas cars and 250 electric cars.
for _ in range(500):
    car = Car('ford', 'focus', 2016)
    gas_fleet.append(car)
for _ in range(250):
    ecar = ElectricCar('nissan', 'leaf', 2016)
    electric_fleet.append(ecar)

# Fill the gas cars, and charge electric cars.
for car in gas_fleet:
    car.fill_tank()
for ecar in electric_fleet:
    ecar.charge()

print("Gas cars:", len(gas_fleet))
print("Electric cars:", len(electric_fleet))
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Files and Exceptions

What are files? What are exceptions?

Your programs can read information in from files, and they can write data to files. Reading from files allows you to work with a wide variety of information; writing to files allows users to pick up where they left off the next time they run your program. You can write text to files, and you can store Python structures such as lists in data files.

Exceptions are special objects that help your programs respond to errors in appropriate ways. For example if your program tries to open a file that doesn't exist, you can use exceptions to display an informative error message instead of having the program crash.

Reading from a file

To read from a file your program needs to open the file and then read the contents of the file. You can read the entire contents of the file at once, or read the file line by line. The with statement makes sure the file is closed properly when the program has finished accessing the file.

Reading an entire file at once

```python
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    contents = f_obj.read()

print(contents)
```

Reading line by line
Each line that's read from the file has a newline character at the end of the line, and the print function adds its own newline character. The rstrip() method gets rid of the the extra blank lines this would result in when printing to the terminal.

```python
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    for line in f_obj:
        print(line.rstrip())
```

Reading from a file (cont.)

Storing the lines in a list

```python
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    lines = f_obj.readlines()

for line in lines:
    print(line.rstrip())
```

Writing to a file
Passing the 'w' argument to open() tells Python you want to write to the file. Be careful; this will erase the contents of the file if it already exists. Passing the 'a' argument tells Python you want to append to the end of an existing file.

Writing to an empty file

```python
filename = 'programming.txt'

with open(filename, 'w') as f:
    f.write("I love programming!")
```

Writing multiple lines to an empty file

```python
filename = 'programming.txt'

with open(filename, 'w') as f:
    f.write("I love programming!\n")
    f.write("I love creating new games.\n")
```

Appending to a file

```python
filename = 'programming.txt'

with open(filename, 'a') as f:
    f.write("I also love working with data.\n")
    f.write("I love making apps as well.\n")
```

File paths
When Python runs the open() function, it looks for the file in the same directory where the program that's being excuted is stored. You can open a file from a subfolder using a relative path. You can also use an absolute path to open any file on your system.

Opening a file from a subfolder

```python
f_path = "text_files/alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()

for line in lines:
    print(line.rstrip())
```

File paths (cont.)

Opening a file using an absolute path

```python
f_path = "/home/ehmatthes/books/alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()
```

Opening a file on Windows
Windows will sometimes interpret forward slashes incorrectly. If you run into this, use backslashes in your file paths.

```python
f_path = "C:\Users\ehmatthes\books\alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()
```

The try-except block
When you think an error may occur, you can write a try-except block to handle the exception that might be raised. The try block tells Python to try running some code, and the except block tells Python what to do if the code results in a particular kind of error.

Handling the ZeroDivisionError exception

```python
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Handling the FileNotFoundError exception

```python
f_name = 'siddhartha.txt'

try:
    with open(f_name) as f_obj:
        lines = f_obj.readlines()
except FileNotFoundError:
    msg = "Can't find file {0}.".format(f_name)
    print(msg)
```

Knowing which exception to handle
It can be hard to know what kind of exception to handle when writing code. Try writing your code without a try block, and make it generate an error. The traceback will tell you what kind of exception your program needs to handle.

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse
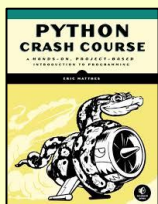
The else block

The try block should only contain code that may cause an error. Any code that depends on the try block running successfully should be placed in the else block.

Using an else block

```python
print("Enter two numbers. I'll divide them.")

x = input("First number: ")
y = input("Second number: ")

try:
    result = int(x) / int(y)
except ZeroDivisionError:
    print("You can't divide by zero!")
else:
    print(result)
```

Preventing crashes from user input
Without the except block in the following example, the program would crash if the user tries to divide by zero. As written, it will handle the error gracefully and keep running.

```python
"""A simple calculator for division only."""

print("Enter two numbers. I'll divide them.")
print("Enter 'q' to quit.")

while True:
    x = input("\nFirst number: ")
    if x == 'q':
        break
    y = input("Second number: ")
    if y == 'q':
        break

    try:
        result = int(x) / int(y)
    except ZeroDivisionError:
        print("You can't divide by zero!")
    else:
        print(result)
```

Deciding which errors to report

Well-written, properly tested code is not very prone to internal errors such as syntax or logical errors. But every time your program depends on something external such as user input or the existence of a file, there's a possibility of an exception being raised.

It's up to you how to communicate errors to your users. Sometimes users need to know if a file is missing; sometimes it's better to handle the error silently. A little experience will help you know how much to report.

Failing silently

Sometimes you want your program to just continue running when it encounters an error, without reporting the error to the user. Using the pass statement in an else block allows you to do this.

Using the pass statement in an else block

```python
f_names = ['alice.txt', 'siddhartha.txt',
           'moby_dick.txt', 'little_women.txt']

for f_name in f_names:
    # Report the length of each file found.
    try:
        with open(f_name) as f_obj:
            lines = f_obj.readlines()
    except FileNotFoundError:
        # Just move on to the next file.
        pass
    else:
        num_lines = len(lines)
        msg = "{0} has {1} lines.".format(
            f_name, num_lines)
        print(msg)
```

Avoid bare except blocks

Exception-handling code should catch specific exceptions that you expect to happen during your program's execution. A bare except block will catch all exceptions, including keyboard interrupts and system exits you might need when forcing a program to close.

If you want to use a try block and you're not sure which exception to catch, use Exception. It will catch most exceptions, but still allow you to interrupt programs intentionally.

Don't use bare except blocks

```python
try:
    # Do something
except:
    pass
```

Use Exception instead

```python
try:
    # Do something
except Exception:
    pass
```

Printing the exception

```python
try:
    # Do something
except Exception as e:
    print(e, type(e))
```

Storing data with json

The json module allows you to dump simple Python data structures into a file, and load the data from that file the next time the program runs. The JSON data format is not specific to Python, so you can share this kind of data with people who work in other languages as well.

Knowing how to manage exceptions is important when working with stored data. You'll usually want to make sure the data you're trying to load exists before working with it.

Using json.dump() to store data

```python
"""Store some numbers."""

import json

numbers = [2, 3, 5, 7, 11, 13]

filename = 'numbers.json'
with open(filename, 'w') as f_obj:
    json.dump(numbers, f_obj)
```

Using json.load() to read data

```python
"""Load some previously stored numbers."""

import json

filename = 'numbers.json'
with open(filename) as f_obj:
    numbers = json.load(f_obj)

print(numbers)
```

Making sure the stored data exists

```python
import json

f_name = 'numbers.json'

try:
    with open(f_name) as f_obj:
        numbers = json.load(f_obj)
except FileNotFoundError:
    msg = "Can't find {0}.".format(f_name)
    print(msg)
else:
    print(numbers)
```

Practice with exceptions
Take a program you've already written that prompts for user input, and add some error-handling code to the program.

Beginner's Python Cheat Sheet — Testing Your Code

Why test your code?

When you write a function or a class, you can also write tests for that code. Testing proves that your code works as it's supposed to in the situations it's designed to handle, and also when people use your programs in unexpected ways. Writing tests gives you confidence that your code will work correctly as more people begin to use your programs. You can also add new features to your programs and know that you haven't broken existing behavior.

A unit test verifies that one specific aspect of your code works as it's supposed to. A test case is a collection of unit tests which verify your code's behavior in a wide variety of situations.

Testing a function: A passing test

Python's unittest module provides tools for testing your code. To try it out, we'll create a function that returns a full name. We'll use the function in a regular program, and then build a test case for the function.

A function to test
Save this as full_names.py

```python
def get_full_name(first, last):
    """Return a full name."""
    full_name = "{0} {1}".format(first, last)
    return full_name.title()
```

Using the function
Save this as names.py

```python
from full_names import get_full_name

janis = get_full_name('janis', 'joplin')
print(janis)

bob = get_full_name('bob', 'dylan')
print(bob)
```

Testing a function (cont.)

Building a testcase with one unit test
To build a test case, make a class that inherits from unittest.TestCase *and write methods that begin with* test_. *Save this as test_full_names.py*

```python
import unittest
from full_names import get_full_name

class NamesTestCase(unittest.TestCase):
    """Tests for names.py."""

    def test_first_last(self):
        """Test names like Janis Joplin."""
        full_name = get_full_name('janis',
                'joplin')
        self.assertEqual(full_name,
                'Janis Joplin')


unittest.main()
```

Running the test
Python reports on each unit test in the test case. The dot reports a single passing test. Python informs us that it ran 1 test in less than 0.001 seconds, and the OK lets us know that all unit tests in the test case passed.

```
.
----------------------------------------
Ran 1 test in 0.000s

OK
```

Testing a function: A failing test

Failing tests are important; they tell you that a change in the code has affected existing behavior. When a test fails, you need to modify the code so the existing behavior still works.

Modifying the function
We'll modify get_full_name() *so it handles middle names, but we'll do it in a way that breaks existing behavior.*

```python
def get_full_name(first, middle, last):
    """Return a full name."""
    full_name = "{0} {1} {2}".format(first,
                                    middle, last)
    return full_name.title()
```

Using the function

```python
from full_names import get_full_name

john = get_full_name('john', 'lee', 'hooker')
print(john)

david = get_full_name('david', 'lee', 'roth')
print(david)
```

A failing test (cont.)

Running the test
When you change your code, it's important to run your existing tests. This will tell you whether the changes you made affected existing behavior.

```
E
==========================================
ERROR: test_first_last (__main__.NamesTestCase)
Test names like Janis Joplin.
------------------------------------------
Traceback (most recent call last):
  File "test_full_names.py", line 10,
    in test_first_last
    'joplin')
TypeError: get_full_name() missing 1 required
  positional argument: 'last'


------------------------------------------
Ran 1 test in 0.001s

FAILED (errors=1)
```

Fixing the code
When a test fails, the code needs to be modified until the test passes again. (Don't make the mistake of rewriting your tests to fit your new code.) Here we can make the middle name optional.

```python
def get_full_name(first, last, middle=''):
    """Return a full name."""
    if middle:
        full_name = "{0} {1} {2}".format(first,
                            middle, last)
    else:
        full_name = "{0} {1}".format(first,
                            last)
    return full_name.title()
```
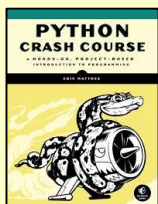
Running the test
Now the test should pass again, which means our original functionality is still intact.

```
.
----------------------------------------
Ran 1 test in 0.000s

OK
```

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse

Adding new tests

You can add as many unit tests to a test case as you need. To write a new test, add a new method to your test case class.

Testing middle names

We've shown that `get_full_name()` *works for first and last names. Let's test that it works for middle names as well.*

```python
import unittest
from full_names import get_full_name

class NamesTestCase(unittest.TestCase):
    """Tests for names.py."""

    def test_first_last(self):
        """Test names like Janis Joplin."""
        full_name = get_full_name('janis',
                'joplin')
        self.assertEqual(full_name,
                'Janis Joplin')

    def test_middle(self):
        """Test names like David Lee Roth."""
        full_name = get_full_name('david',
                'roth', 'lee')
        self.assertEqual(full_name,
                'David Lee Roth')

unittest.main()
```

Running the tests

The two dots represent two passing tests.

```
..
----------------------------------
Ran 2 tests in 0.000s

OK
```

A variety of assert methods

Python provides a number of assert methods you can use to test your code.

Verify that a==b, or a != b

```python
assertEqual(a, b)
assertNotEqual(a, b)
```

Verify that x is True, or x is False

```python
assertTrue(x)
assertFalse(x)
```

Verify an item is in a list, or not in a list

```python
assertIn(item, list)
assertNotIn(item, list)
```

Testing a class

Testing a class is similar to testing a function, since you'll mostly be testing your methods.

A class to test
Save as accountant.py

```python
class Accountant():
    """Manage a bank account."""

    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount
```

Building a testcase

For the first test, we'll make sure we can start out with different initial balances. Save this as test_accountant.py.

```python
import unittest
from accountant import Accountant

class TestAccountant(unittest.TestCase):
    """Tests for the class Accountant."""

    def test_initial_balance(self):
        # Default balance should be 0.
        acc = Accountant()
        self.assertEqual(acc.balance, 0)

        # Test non-default balance.
        acc = Accountant(100)
        self.assertEqual(acc.balance, 100)

unittest.main()
```

Running the test

```
.
----------------------------------------
Ran 1 test in 0.000s

OK
```

When is it okay to modify tests?

In general you shouldn't modify a test once it's written. When a test fails it usually means new code you've written has broken existing functionality, and you need to modify the new code until all existing tests pass.

If your original requirements have changed, it may be appropriate to modify some tests. This usually happens in the early stages of a project when desired behavior is still being sorted out.

The setUp() method

When testing a class, you usually have to make an instance of the class. The `setUp()` *method is run before every test. Any instances you make in* `setUp()` *are available in every test you write.*

Using setUp() to support multiple tests
The instance `self.acc` *can be used in each new test.*

```python
import unittest
from accountant import Accountant

class TestAccountant(unittest.TestCase):
    """Tests for the class Accountant."""

    def setUp(self):
        self.acc = Accountant()

    def test_initial_balance(self):
        # Default balance should be 0.
        self.assertEqual(self.acc.balance, 0)

        # Test non-default balance.
        acc = Accountant(100)
        self.assertEqual(acc.balance, 100)

    def test_deposit(self):
        # Test single deposit.
        self.acc.deposit(100)
        self.assertEqual(self.acc.balance, 100)

        # Test multiple deposits.
        self.acc.deposit(100)
        self.acc.deposit(100)
        self.assertEqual(self.acc.balance, 300)

    def test_withdrawal(self):
        # Test single withdrawal.
        self.acc.deposit(1000)
        self.acc.withdraw(100)
        self.assertEqual(self.acc.balance, 900)

unittest.main()
```

Running the tests

```
...
----------------------------------
Ran 3 tests in 0.001s

OK
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet - Pygame

What is Pygame?

Pygame is a framework for making games using Python. Making games is fun, and it's a great way to expand your programming skills and knowledge. Pygame takes care of many of the lower-level tasks in building games, which lets you focus on the aspects of your game that make it interesting.

Installing Pygame

Pygame runs on all systems, but setup is slightly different on each OS. The instructions here assume you're using Python 3, and provide a minimal installation of Pygame. If these instructions don't work for your system, see the more detailed notes at http://ehmatthes.github.io/pcc/.

Pygame on Linux

```
$ sudo apt-get install python3-dev mercurial
    libsdl-image1.2-dev libsdl2-dev
    libsdl-ttf2.0-dev
$ pip install --user
    hg+http://bitbucket.org/pygame/pygame
```

Pygame on OS X
This assumes you've used Homebrew to install Python 3.

```
$ brew install hg sdl sdl_image sdl_ttf
$ pip install --user
    hg+http://bitbucket.org/pygame/pygame
```

Pygame on Windows
Find an installer at https://bitbucket.org/pygame/pygame/downloads/ or http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygame that matches your version of Python. Run the installer file if it's a .exe or .msi file. If it's a .whl file, use pip to install Pygame:

```
> python -m pip install --user
    pygame-1.9.2a0-cp35-none-win32.whl
```

Testing your installation
To test your installation, open a terminal session and try to import Pygame. If you don't get any error messages, your installation was successful.

```
$ python
>>> import pygame
>>>
```

Starting a game
The following code sets up an empty game window, and starts an event loop and a loop that continually refreshes the screen.

An empty game window

```python
import sys
import pygame as pg

def run_game():
    # Initialize and set up screen.
    pg.init()
    screen = pg.display.set_mode((1200, 800))
    pg.display.set_caption("Alien Invasion")

    # Start main loop.
    while True:
        # Start event loop.
        for event in pg.event.get():
            if event.type == pg.QUIT:
                sys.exit()

        # Refresh screen.
        pg.display.flip()

run_game()
```

Setting a custom window size
The display.set_mode() function accepts a tuple that defines the screen size.

```python
screen_dim = (1200, 800)
screen = pg.display.set_mode(screen_dim)
```

Setting a custom background color
Colors are defined as a tuple of red, green, and blue values. Each value ranges from 0-255.

```python
bg_color = (230, 230, 230)
screen.fill(bg_color)
```

Pygame rect objects

Many objects in a game can be treated as simple rectangles, rather than their actual shape. This simplifies code without noticeably affecting game play. Pygame has a rect object that makes it easy to work with game objects.

Getting the screen rect object
We already have a screen object; we can easily access the rect object associated with the screen.

```python
screen_rect = screen.get_rect()
```

Finding the center of the screen
Rect objects have a center attribute which stores the center point.

```python
screen_center = screen_rect.center
```

Pygame rect objects (cont.)

Useful rect attributes
Once you have a rect object, there are a number of attributes that are useful when positioning objects and detecting relative positions of objects. (You can find more attributes in the Pygame documentation.)

```python
# Individual x and y values:
screen_rect.left, screen_rect.right
screen_rect.top, screen_rect.bottom
screen_rect.centerx, screen_rect.centery
screen_rect.width, screen_rect.height

# Tuples
screen_rect.center
screen_rect.size
```

Creating a rect object
You can create a rect object from scratch. For example a small rect object that's filled in can represent a bullet in a game. The Rect() class takes the coordinates of the upper left corner, and the width and height of the rect. The draw.rect() function takes a screen object, a color, and a rect. This function fills the given rect with the given color.

```python
bullet_rect = pg.Rect(100, 100, 3, 15)
color = (100, 100, 100)
pg.draw.rect(screen, color, bullet_rect)
```

Working with images

Many objects in a game are images that are moved around the screen. It's easiest to use bitmap (.bmp) image files, but you can also configure your system to work with jpg, png, and gif files as well.

Loading an image

```python
ship = pg.image.load('images/ship.bmp')
```

Getting the rect object from an image

```python
ship_rect = ship.get_rect()
```
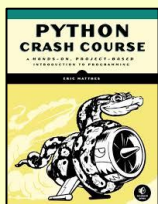
Positioning an image
With rects, it's easy to position an image wherever you want on the screen, or in relation to another object. The following code positions a ship object at the bottom center of the screen.

```python
ship_rect.midbottom = screen_rect.midbottom
```

Working with images (cont.)

Drawing an image to the screen
Once an image is loaded and positioned, you can draw it to the screen with the blit() method. The blit() method acts on the screen object, and takes the image object and image rect as arguments.

```python
# Draw ship to screen.
screen.blit(ship, ship_rect)
```

The blitme() method
Game objects such as ships are often written as classes. Then a blitme() method is usually defined, which draws the object to the screen.

```python
def blitme(self):
    """Draw ship at current location."""
    self.screen.blit(self.image, self.rect)
```

Responding to keyboard input
Pygame watches for events such as key presses and mouse actions. You can detect any event you care about in the event loop, and respond with any action that's appropriate for your game.

Responding to key presses
Pygame's main event loop registers a KEYDOWN event any time a key is pressed. When this happens, you can check for specific keys.

```python
for event in pg.event.get():
    if event.type == pg.KEYDOWN:
        if event.key == pg.K_RIGHT:
            ship_rect.x += 1
        elif event.key == pg.K_LEFT:
            ship_rect.x -= 1
        elif event.key == pg.K_SPACE:
            ship.fire_bullet()
        elif event.key == pg.K_q:
            sys.exit()
```

Responding to released keys
When the user releases a key, a KEYUP event is triggered.

```python
if event.type == pg.KEYUP:
    if event.key == pg.K_RIGHT:
        ship.moving_right = False
```

Pygame documentation
The Pygame documentation is really helpful when building your own games. The home page for the Pygame project is at http://pygame.org/, and the home page for the documentation is at http://pygame.org/docs/.

The most useful part of the documentation are the pages about specific parts of Pygame, such as the Rect() class and the sprite module. You can find a list of these elements at the top of the help pages.

Responding to mouse events
Pygame's event loop registers an event any time the mouse moves, or a mouse button is pressed or released.

Responding to the mouse button

```python
for event in pg.event.get():
    if event.type == pg.MOUSEBUTTONDOWN:
        ship.fire_bullet()
```

Finding the mouse position
The mouse position is returned as a tuple.

```python
mouse_pos = pg.mouse.get_pos()
```

Clicking a button
You might want to know if the cursor is over an object such as a button. The rect.collidepoint() method returns true when a point is inside a rect object.

```python
if button_rect.collidepoint(mouse_pos):
    start_game()
```

Hiding the mouse

```python
pg.mouse.set_visible(False)
```

Pygame groups
Pygame has a Group class which makes working with a group of similar objects easier. A group is like a list, with some extra functionality that's helpful when building games.

Making and filling a group
An object that will be placed in a group must inherit from Sprite.

```python
from pygame.sprite import Sprite, Group

def Bullet(Sprite):
    ...
    def draw_bullet(self):
        ...
    def update(self):
        ...

bullets = Group()

new_bullet = Bullet()
bullets.add(new_bullet)
```

Looping through the items in a group
The sprites() method returns all the members of a group.

```python
for bullet in bullets.sprites():
    bullet.draw_bullet()
```

Calling update() on a group
Calling update() on a group automatically calls update() on each member of the group.

```python
bullets.update()
```

Pygame groups (cont.)

Removing an item from a group
It's important to delete elements that will never appear again in the game, so you don't waste memory and resources.

```python
bullets.remove(bullet)
```

Detecting collisions
You can detect when a single object collides with any member of a group. You can also detect when any member of one group collides with a member of another group.

Collisions between a single object and a group
The spritecollideany() function takes an object and a group, and returns True if the object overlaps with any member of the group.

```python
if pg.sprite.spritecollideany(ship, aliens):
    ships_left -= 1
```

Collisions between two groups
The sprite.groupcollide() function takes two groups, and two booleans. The function returns a dictionary containing information about the members that have collided. The booleans tell Pygame whether to delete the members of either group that have collided.

```python
collisions = pg.sprite.groupcollide(
        bullets, aliens, True, True)

score += len(collisions) * alien_point_value
```

Rendering text
You can use text for a variety of purposes in a game. For example you can share information with players, and you can display a score.

Displaying a message
The following code defines a message, then a color for the text and the background color for the message. A font is defined using the default system font, with a font size of 48. The font.render() function is used to create an image of the message, and we get the rect object associated with the image. We then center the image on the screen and display it.

```python
msg = "Play again?"
msg_color = (100, 100, 100)
bg_color = (230, 230, 230)

f = pg.font.SysFont(None, 48)
msg_image = f.render(msg, True, msg_color,
        bg_color)
msg_image_rect = msg_image.get_rect()
msg_image_rect.center = screen_rect.center
screen.blit(msg_image, msg_image_rect)
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — matplotlib

What is matplotlib?

Data visualization involves exploring data through visual representations. The matplotlib package helps you make visually appealing representations of the data you're working with. matplotlib is extremely flexible; these examples will help you get started with a few simple visualizations.

Installing matplotlib

matplotlib runs on all systems, but setup is slightly different depending on your OS. If the minimal instructions here don't work for you, see the more detailed instructions at http://ehmatthes.github.io/pcc/. You should also consider installing the Anaconda distrubution of Python from https://continuum.io/downloads/, which includes matplotlib.

matplotlib on Linux

```
$ sudo apt-get install python3-matplotlib
```

matplotlib on OS X

Start a terminal session and enter `import matplotlib` *to see if it's already installed on your system. If not, try this command:*

```
$ pip install --user matplotlib
```

matplotlib on Windows

You first need to install Visual Studio, which you can do from https://dev.windows.com/. The Community edition is free. Then go to https://pypi.python.org/pypi/matplotlib/ or http://www.lfd.uic.edu/~gohlke/pythonlibs/#matplotlib and download an appropriate installer file.

Line graphs and scatter plots

Making a line graph

```python
import matplotlib.pyplot as plt

x_values = [0, 1, 2, 3, 4, 5]
squares = [0, 1, 4, 9, 16, 25]
plt.plot(x_values, squares)
plt.show()
```

Line graphs and scatter plots (cont.)

Making a scatter plot

The scatter() function takes a list of x values and a list of y values, and a variety of optional arguments. The `s=10` *argument controls the size of each point.*

```python
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]

plt.scatter(x_values, squares, s=10)
plt.show()
```

Customizing plots

Plots can be customized in a wide variety of ways. Just about any element of a plot can be customized.

Adding titles and labels, and scaling axes

```python
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]
plt.scatter(x_values, squares, s=10)

plt.title("Square Numbers", fontsize=24)
plt.xlabel("Value", fontsize=18)
plt.ylabel("Square of Value", fontsize=18)
plt.tick_params(axis='both', which='major',
                    labelsize=14)
plt.axis([0, 1100, 0, 1100000])

plt.show()
```

Using a colormap

A colormap varies the point colors from one shade to another, based on a certain value for each point. The value used to determine the color of each point is passed to the `c` *argument, and the* `cmap` *argument specifies which colormap to use.*
The `edgecolor='none'` *argument removes the black outline from each point.*

```python
plt.scatter(x_values, squares, c=squares,
        cmap=plt.cm.Blues, edgecolor='none',
        s=10)
```
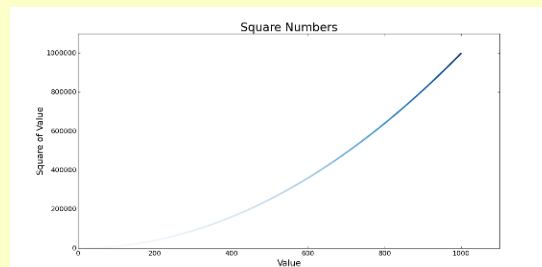

Customizing plots (cont.)

Emphasizing points

You can plot as much data as you want on one plot. Here we re-plot the first and last points larger to emphasize them.

```python
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]
plt.scatter(x_values, squares, c=squares,
        cmap=plt.cm.Blues, edgecolor='none',
        s=10)

plt.scatter(x_values[0], squares[0], c='green',
        edgecolor='none', s=100)
plt.scatter(x_values[-1], squares[-1], c='red',
        edgecolor='none', s=100)

plt.title("Square Numbers", fontsize=24)
--snip--
```

Removing axes

You can customize or remove axes entirely. Here's how to access each axis, and hide it.

```python
plt.axes().get_xaxis().set_visible(False)
plt.axes().get_yaxis().set_visible(False)
```

Setting a custom figure size

You can make your plot as big or small as you want. Before plotting your data, add the following code. The `dpi` *argument is optional; if you don't know your system's resolution you can omit the argument and adjust the* `figsize` *argument accordingly.*

```python
plt.figure(dpi=128, figsize=(10, 6))
```

Saving a plot

The matplotlib viewer has an interactive save button, but you can also save your visualizations programmatically. To do so, replace `plt.show()` *with* `plt.savefig()`. *The* `bbox_inches='tight'` *argument trims extra whitespace from the plot.*

```python
plt.savefig('squares.png', bbox_inches='tight')
```
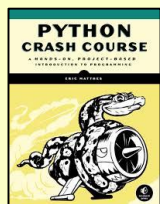
Online resources

The matplotlib gallery and documentation are at http://matplotlib.org/. Be sure to visit the examples, gallery, and pyplot links.

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse

Multiple plots

You can make as many plots as you want on one figure. When you make multiple plots, you can emphasize relationships in the data. For example you can fill the space between two sets of data.

Plotting two sets of data
Here we use `plt.scatter()` *twice to plot square numbers and cubes on the same figure.*

```python
import matplotlib.pyplot as plt

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

plt.scatter(x_values, squares, c='blue',
        edgecolor='none', s=20)
plt.scatter(x_values, cubes, c='red',
        edgecolor='none', s=20)

plt.axis([0, 11, 0, 1100])
plt.show()
```

Filling the space between data sets
The `fill_between()` *method fills the space between two data sets. It takes a series of x-values and two series of y-values. It also takes a* `facecolor` *to use for the fill, and an optional* `alpha` *argument that controls the color's transparency.*

```python
plt.fill_between(x_values, cubes, squares,
        facecolor='blue', alpha=0.25)
```

Working with dates and times

Many interesting data sets have a date or time as the x-value. Python's datetime module helps you work with this kind of data.

Generating the current date
The `datetime.now()` *function returns a datetime object representing the current date and time.*

```python
from datetime import datetime as dt

today = dt.now()
date_string = dt.strftime(today, '%m/%d/%Y')
print(date_string)
```

Generating a specific date
You can also generate a datetime object for any date and time you want. The positional order of arguments is year, month, and day. The hour, minute, second, and microsecond arguments are optional.

```python
from datetime import datetime as dt

new_years = dt(2017, 1, 1)
fall_equinox = dt(year=2016, month=9, day=22)
```

Working with dates and times (cont.)

Datetime formatting arguments
The `strftime()` *function generates a formatted string from a datetime object, and the* `strptime()` *function genereates a datetime object from a string. The following codes let you work with dates exactly as you need to.*

%A	Weekday name, such as Monday
%B	Month name, such as January
%m	Month, as a number (01 to 12)
%d	Day of the month, as a number (01 to 31)
%Y	Four-digit year, such as 2016
%y	Two-digit year, such as 16
%H	Hour, in 24-hour format (00 to 23)
%I	Hour, in 12-hour format (01 to 12)
%p	AM or PM
%M	Minutes (00 to 59)
%S	Seconds (00 to 61)

Converting a string to a datetime object

```python
new_years = dt.strptime('1/1/2017', '%m/%d/%Y')
```

Converting a datetime object to a string

```python
ny_string = dt.strftime(new_years, '%B %d, %Y')
print(ny_string)
```

Plotting high temperatures
The following code creates a list of dates and a corresponding list of high temperatures. It then plots the high temperatures, with the date labels displayed in a specific format.

```python
from datetime import datetime as dt

import matplotlib.pyplot as plt
from matplotlib import dates as mdates

dates = [
    dt(2016, 6, 21), dt(2016, 6, 22),
    dt(2016, 6, 23), dt(2016, 6, 24),
    ]

highs = [57, 68, 64, 59]

fig = plt.figure(dpi=128, figsize=(10,6))
plt.plot(dates, highs, c='red')
plt.title("Daily High Temps", fontsize=24)
plt.ylabel("Temp (F)", fontsize=16)

x_axis = plt.axes().get_xaxis()
x_axis.set_major_formatter(
        mdates.DateFormatter('%B %d %Y')
        )
fig.autofmt_xdate()

plt.show()
```

Multiple plots in one figure

You can include as many individual graphs in one figure as you want. This is useful, for example, when comparing related datasets.

Sharing an x-axis
The following code plots a set of squares and a set of cubes on two separate graphs that share a common x-axis.

The `plt.subplots()` *function returns a figure object and a tuple of axes. Each set of axes corresponds to a separate plot in the figure. The first two arguments control the number of rows and columns generated in the figure.*
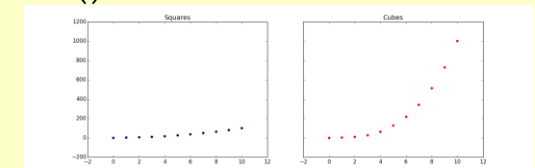
```python
import matplotlib.pyplot as plt

x_vals = list(range(11))
squares = [x**2 for x in x_vals]
cubes = [x**3 for x in x_vals]

fig, axarr = plt.subplots(2, 1, sharex=True)

axarr[0].scatter(x_vals, squares)
axarr[0].set_title('Squares')

axarr[1].scatter(x_vals, cubes, c='red')
axarr[1].set_title('Cubes')

plt.show()
```

Sharing a y-axis
To share a y-axis, we use the `sharey=True` *argument.*

```python
import matplotlib.pyplot as plt

x_vals = list(range(11))
squares = [x**2 for x in x_vals]
cubes = [x**3 for x in x_vals]

fig, axarr = plt.subplots(1, 2, sharey=True)

axarr[0].scatter(x_vals, squares)
axarr[0].set_title('Squares')

axarr[1].scatter(x_vals, cubes, c='red')
axarr[1].set_title('Cubes')

plt.show()
```


More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Pygal

What is Pygal?

Data visualization involves exploring data through visual representations. Pygal helps you make visually appealing representations of the data you're working with. Pygal is particularly well suited for visualizations that will be presented online, because it supports interactive elements.

Installing Pygal
Pygal can be installed using pip.

Pygal on Linux and OS X

```
$ pip install --user pygal
```

Pygal on Windows

```
> python –m pip install --user pygal
```

Line graphs, scatter plots, and bar graphs
To make a plot with Pygal, you specify the kind of plot and then add the data.

Making a line graph
To view the output, open the file squares.svg in a browser.

```
import pygal

x_values = [0, 1, 2, 3, 4, 5]
squares = [0, 1, 4, 9, 16, 25]

chart = pygal.Line()
chart.force_uri_protocol = 'http'
chart.add('x^2', squares)
chart.render_to_file('squares.svg')
```

Adding labels and a title

```
--snip--
chart = pygal.Line()
chart.force_uri_protocol = 'http'
chart.title = "Squares"
chart.x_labels = x_values
chart.x_title = "Value"
chart.y_title = "Square of Value"
chart.add('x^2', squares)
chart.render_to_file('squares.svg')
```

Line graphs, scatter plots, and bar graphs (cont.)

Making a scatter plot
The data for a scatter plot needs to be a list containing tuples of the form (x, y). The stroke=False *argument tells Pygal to make an XY chart with no line connecting the points.*

```
import pygal

squares = [
    (0, 0), (1, 1), (2, 4), (3, 9),
    (4, 16), (5, 25),
    ]

chart = pygal.XY(stroke=False)
chart.force_uri_protocol = 'http'
chart.add('x^2', squares)
chart.render_to_file('squares.svg')
```

Using a list comprehension for a scatter plot
A list comprehension can be used to effficiently make a dataset for a scatter plot.

```
squares = [(x, x**2) for x in range(1000)]
```

Making a bar graph
A bar graph requires a list of values for the bar sizes. To label the bars, pass a list of the same length to x_labels.

```
import pygal

outcomes = [1, 2, 3, 4, 5, 6]
frequencies = [18, 16, 18, 17, 18, 13]

chart = pygal.Bar()
chart.force_uri_protocol = 'http'
chart.x_labels = outcomes
chart.add('D6', frequencies)
chart.render_to_file('rolling_dice.svg')
```

Making a bar graph from a dictionary
Since each bar needs a label and a value, a dictionary is a great way to store the data for a bar graph. The keys are used as the labels along the x-axis, and the values are used to determine the height of each bar.

```
import pygal

results = {
    1:18, 2:16, 3:18,
    4:17, 5:18, 6:13,
    }

chart = pygal.Bar()
chart.force_uri_protocol = 'http'
chart.x_labels = results.keys()
chart.add('D6', results.values())
chart.render_to_file('rolling_dice.svg')
```

Multiple plots
You can add as much data as you want when making a visualization.

Plotting squares and cubes

```
import pygal

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

chart = pygal.Line()
chart.force_uri_protocol = 'http'
chart.title = "Squares and Cubes"
chart.x_labels = x_values

chart.add('Squares', squares)
chart.add('Cubes', cubes)
chart.render_to_file('squares_cubes.svg')
```
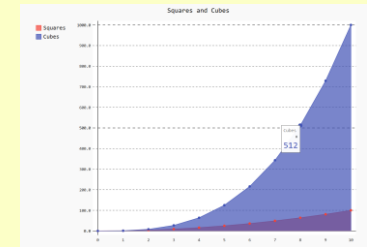
Filling the area under a data series
Pygal allows you to fill the area under or over each series of data. The default is to fill from the x-axis up, but you can fill from any horizontal line using the zero *argument.*

```
chart = pygal.Line(fill=True, zero=0)
```


Online resources
The documentation for Pygal is available at http://www.pygal.org/.
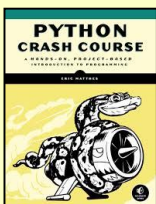
Enabling interactive features
If you're viewing svg output in a browser, Pygal needs to render the output file in a specific way. The force_uri_protocol *attribute for chart objects needs to be set to* 'http'.

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse

Styling plots

Pygal lets you customize many elements of a plot. There are some excellent default themes, and many options for styling individual plot elements.

Using built-in styles
To use built-in styles, import the style and make an instance of the style class. Then pass the style object with the style *argument when you make the chart object.*

```
import pygal
from pygal.style import LightGreenStyle

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

chart_style = LightGreenStyle()
chart = pygal.Line(style=chart_style)
chart.force_uri_protocol = 'http'
chart.title = "Squares and Cubes"
chart.x_labels = x_values

chart.add('Squares', squares)
chart.add('Cubes', cubes)
chart.render_to_file('squares_cubes.svg')
```

Parametric built-in styles
Some built-in styles accept a custom color, then generate a theme based on that color.

```
from pygal.style import LightenStyle

--snip--
chart_style = LightenStyle('#336688')
chart = pygal.Line(style=chart_style)
--snip--
```

Customizing individual style properties
Style objects have a number of properties you can set individually.

```
chart_style = LightenStyle('#336688')
chart_style.plot_background = '#CCCCCC'
chart_style.major_label_font_size = 20
chart_style.label_font_size = 16
--snip--
```

Custom style class
You can start with a bare style class, and then set only the properties you care about.

```
chart_style = Style()
chart_style.colors = [
    '#CCCCCC', '#AAAAAA', '#888888']
chart_style.plot_background = '#EEEEEE'

chart = pygal.Line(style=chart_style)
--snip--
```

Styling plots (cont.)

Configuration settings
Some settings are controlled by a Config *object.*

```
my_config = pygal.Config()
my_config.show_y_guides = False
my_config.width = 1000
my_config.dots_size = 5

chart = pygal.Line(config=my_config)
--snip--
```

Styling series
You can give each series on a chart different style settings.

```
chart.add('Squares', squares, dots_size=2)
chart.add('Cubes', cubes, dots_size=3)
```

Styling individual data points
You can style individual data points as well. To do so, write a dictionary for each data point you want to customize. A 'value' *key is required, and other properies are optional.*

```
import pygal

repos = [
    {
        'value': 20506,
        'color': '#3333CC',
        'xlink': 'http://djangoproject.com/',
    },
    20054,
    12607,
    11827,
    ]

chart = pygal.Bar()
chart.force_uri_protocol = 'http'
chart.x_labels = [
    'django', 'requests', 'scikit-learn',
    'tornado',
    ]
chart.y_title = 'Stars'
chart.add('Python Repos', repos)
chart.render_to_file('python_repos.svg')
```
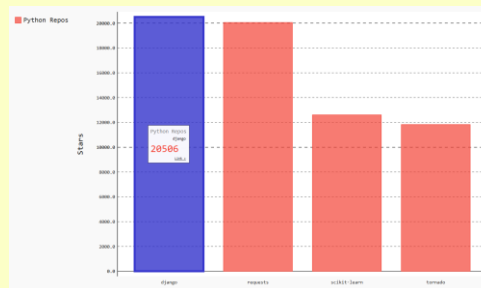

Plotting global datasets

Pygal can generate world maps, and you can add any data you want to these maps. Data is indicated by coloring, by labels, and by tooltips that show data when users hover over each country on the map.

Installing the world map module
The world map module is not included by default in Pygal 2.0. It can be installed with pip:

```
$ pip install --user pygal_maps_world
```

Making a world map
The following code makes a simple world map showing the countries of North America.

```
from pygal.maps.world import World

wm = World()
wm.force_uri_protocol = 'http'
wm.title = 'North America'
wm.add('North America', ['ca', 'mx', 'us'])

wm.render_to_file('north_america.svg')
```

Showing all the country codes
In order to make maps, you need to know Pygal's country codes. The following example will print an alphabetical list of each country and its code.

```
from pygal.maps.world import COUNTRIES

for code in sorted(COUNTRIES.keys()):
    print(code, COUNTRIES[code])
```

Plotting numerical data on a world map
To plot numerical data on a map, pass a dictionary to add() *instead of a list.*

```
from pygal.maps.world import World

populations = {
    'ca': 34126000,
    'us': 309349000,
    'mx': 113423000,
    }

wm = World()
wm.force_uri_protocol = 'http'
wm.title = 'Population of North America'
wm.add('North America', populations)

wm.render_to_file('na_populations.svg')
```

Beginner's Python Cheat Sheet — Django

What is Django?

Django is a web framework which helps you build interactive websites using Python. With Django you define the kind of data your site needs to work with, and you define the ways your users can work with that data.

Installing Django

It's usualy best to install Django to a virtual environment, where your project can be isolated from your other Python projects. Most commands assume you're working in an active virtual environment.

Create a virtual environment

```
$ python -m venv ll_env
```

Activate the environment (Linux and OS X)

```
$ source ll_env/bin/activate
```

Activate the environment (Windows)

```
> ll_env\Scripts\activate
```

Install Django to the active environment

```
(ll_env)$ pip install Django
```

Creating a project

To start a project we'll create a new project, create a database, and start a development server.

Create a new project

```
$ django-admin.py startproject learning_log .
```

Create a database

```
$ python manage.py migrate
```

View the project

After issuing this command, you can view the project at http://localhost:8000/.

```
$ python manage.py runserver
```

Create a new app

A Django project is made up of one or more apps.

```
$ python manage.py startapp learning_logs
```

Working with models

The data in a Django project is structured as a set of models.

Defining a model

To define the models for your app, modify the file models.py that was created in your app's folder. The __str__() method tells Django how to represent data objects based on this model.

```
from django.db import models

class Topic(models.Model):
    """A topic the user is learning about."""
    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(
            auto_now_add=True)

    def __str__(self):
        return self.text
```

Activating a model

To use a model the app must be added to the tuple INSTALLED_APPS, which is stored in the project's settings.py file.

```
INSTALLED_APPS = (
    --snip--
    'django.contrib.staticfiles',

    # My apps
    'learning_logs',
)
```

Migrating the database

The database needs to be modified to store the kind of data that the model represents.

```
$ python manage.py makemigrations learning_logs
$ python manage.py migrate
```

Creating a superuser

A superuser is a user account that has access to all aspects of the project.

```
$ python manage.py createsuperuser
```

Registering a model

You can register your models with Django's admin site, which makes it easier to work with the data in your project. To do this, modify the app's admin.py file. View the admin site at http://localhost:8000/admin/.

```
from django.contrib import admin

from learning_logs.models import Topic

admin.site.register(Topic)
```

Building a simple home page

Users interact with a project through web pages, and a project's home page can start out as a simple page with no data. A page usually needs a URL, a view, and a template.

Mapping a project's URLs

The project's main urls.py file tells Django where to find the urls.py files associated with each app in the project.

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'', include('learning_logs.urls',
        namespace='learning_logs')),
    ]
```

Mapping an app's URLs

An app's urls.py file tells Django which view to use for each URL in the app. You'll need to make this file yourself, and save it in the app's folder.

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    ]
```

Writing a simple view

A view takes information from a request and sends data to the browser, often through a template. View functions are stored in an app's views.py file. This simple view function doesn't pull in any data, but it uses the template index.html to render the home page.

```
from django.shortcuts import render

def index(request):
    """The home page for Learning Log."""
    return render(request,
            'learning_logs/index.html')
```
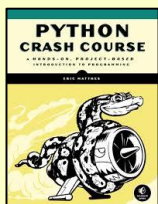
Online resources

The documentation for Django is available at http://docs.djangoproject.com/. The Django documentation is thorough and user-friendly, so check it out!

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse

Building a simple home page (cont.)

Writing a simple template
A template sets up the structure for a page. It's a mix of html and template code, which is like Python but not as powerful. Make a folder called templates inside the project folder. Inside the templates folder make another folder with the same name as the app. This is where the template files should be saved.

```
<p>Learning Log</p>

<p>Learning Log helps you keep track of your
learning, for any topic you're learning
about.</p>
```

Template inheritance
Many elements of a web page are repeated on every page in the site, or every page in a section of the site. By writing one parent template for the site, and one for each section, you can easily modify the look and feel of your entire site.

The parent template
The parent template defines the elements common to a set of pages, and defines blocks that will be filled by individual pages.

```
<p>
  <a href="{% url 'learning_logs:index' %}">
    Learning Log
  </a>
</p>

{% block content %}{% endblock content %}
```

The child template
The child template uses the {% extends %} template tag to pull in the structure of the parent template. It then defines the content for any blocks defined in the parent template.

```
{% extends 'learning_logs/base.html' %}

{% block content %}
  <p>
    Learning Log helps you keep track
    of your learning, for any topic you're
    learning about.
  </p>
{% endblock content %}
```

Template indentation
Python code is usually indented by four spaces. In templates you'll often see two spaces used for indentation, because elements tend to be nested more deeply in templates.

Another model
A new model can use an existing model. The ForeignKey attribute establishes a connection between instances of the two related models. Make sure to migrate the database after adding a new model to your app.

Defining a model with a foreign key

```
class Entry(models.Model):
    """Learning log entries for a topic."""
    topic = models.ForeignKey(Topic)
    text = models.TextField()
    date_added = models.DateTimeField(
            auto_now_add=True)

    def __str__(self):
        return self.text[:50] + "..."
```

Building a page with data
Most pages in a project need to present data that's specific to the current user.

URL parameters
A URL often needs to accept a parameter telling it which data to access from the database. The second URL pattern shown here looks for the ID of a specific topic and stores it in the parameter topic_id.

```
urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^topics/(?P<topic_id>\d+)/$',
            views.topic, name='topic'),
]
```

Using data in a view
The view uses a parameter from the URL to pull the correct data from the database. In this example the view is sending a context dictionary to the template, containing data that should be displayed on the page.

```
def topic(request, topic_id):
    """Show a topic and all its entries."""
    topic = Topics.objects.get(id=topic_id)
    entries = topic.entry_set.order_by(
            '-date_added')
    context = {
        'topic': topic,
        'entries': entries,
        }
    return render(request,
        'learning_logs/topic.html', context)
```

Restarting the development server
If you make a change to your project and the change doesn't seem to have any effect, try restarting the server:
```
$ python manage.py runserver
```

Building a page with data (cont.)

Using data in a template
The data in the view function's context dictionary is available within the template. This data is accessed using template variables, which are indicated by doubled curly braces.
* The vertical line after a template variable indicates a filter. In this case a filter called date formats date objects, and the filter linebreaks renders paragraphs properly on a web page.*

```
{% extends 'learning_logs/base.html' %}

{% block content %}

  <p>Topic: {{ topic }}</p>

  <p>Entries:</p>
  <ul>
  {% for entry in entries %}
    <li>
    <p>
      {{ entry.date_added|date:'M d, Y H:i' }}
    </p>
    <p>
      {{ entry.text|linebreaks }}
    </p>
    </li>
  {% empty %}
    <li>There are no entries yet.</li>
  {% endfor %}
  </ul>

{% endblock content %}
```

The Django shell
You can explore the data in your project from the command line. This is helpful for developing queries and testing code snippets.

Start a shell session
```
$ python manage.py shell
```

Access data from the project
```
>>> from learning_logs.models import Topic
>>> Topic.objects.all()
[<Topic: Chess>, <Topic: Rock Climbing>]
>>> topic = Topic.objects.get(id=1)
>>> topic.text
'Chess'
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Django, Part 2

Users and forms

Most web applications need to let users create accounts. This lets users create and work with their own data. Some of this data may be private, and some may be public. Django's forms allow users to enter and modify their data.

User accounts

User accounts are handled by a dedicated app called users. *Users need to be able to register, log in, and log out. Django automates much of this work for you.*

Making a users app
After making the app, be sure to add 'users' *to* INSTALLED_APPS *in the project's settings.py file.*

```
$ python manage.py startapp users
```

Including URLS for the users app
Add a line to the project's urls.py file so the users *app's URLs are included in the project.*

```
urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^users/', include('users.urls',
            namespace='users')),
    url(r'', include('learning_logs.urls',
            namespace='learning_logs')),
    ]
```

Using forms in Django

There are a number of ways to create forms and work with them. You can use Django's defaults, or completely customize your forms. For a simple way to let users enter data based on your models, use a ModelForm. *This creates a form that allows users to enter data that will populate the fields on a model.*

 The register view on the back of this sheet shows a simple approach to form processing. If the view doesn't receive data from a form, it responds with a blank form. If it receives POST *data from a form, it validates the data and then saves it to the database.*

Defining the URLs
Users will need to be able to log in, log out, and register. Make a new urls.py file in the users *app folder. The login view is a default view provided by Django.*

```
from django.conf.urls import url
from django.contrib.auth.views import login

from . import views

urlpatterns = [
    url(r'^login/$', login,
        {'template_name': 'users/login.html'},
        name='login'),
    url(r'^logout/$', views.logout_view,
        name='logout'),
    url(r'^register/$', views.register,
        name='register'),
    ]
```

The login template
The login view is provided by default, but you need to provide your own login template. The template shown here displays a simple login form, and provides basic error messages. Make a templates folder in the users folder, and then make a users folder in the templates folder. Save this file as login.html.

 The tag {% csrf_token %} *helps prevent a common type of attack with forms. The* {{ form.as_p }} *element displays the default login form in paragraph format. The* <input> *element named* next *redirects the user to the home page after a successful login.*

```
{% extends "learning_logs/base.html" %}

{% block content %}
  {% if form.errors %}
  <p>
    Your username and password didn't match.
    Please try again.
  </p>
  {% endif %}

  <form method="post"
      action="{% url 'users:login' %}">
    {% csrf token %}
    {{ form.as_p }}
    <button name="submit">log in</button>

    <input type="hidden" name="next"
      value="{% url 'learning_logs:index' %}"/>
  </form>

{% endblock content %}
```

Showing the current login status
You can modify the base.html template to show whether the user is currently logged in, and to provide a link to the login and logout pages. Django makes a user *object available to every template, and this template takes advantage of this object.*

 The user.is_authenticated *tag allows you to serve specific content to users depending on whether they have logged in or not. The* {{ user.username }} *property allows you to greet users who have logged in. Users who haven't logged in see links to register or log in.*

```
<p>
  <a href="{% url 'learning_logs:index' %}">
    Learning Log
  </a>
  {% if user.is_authenticated %}
    Hello, {{ user.username }}.
    <a href="{% url 'users:logout' %}">
      log out
    </a>
  {% else %}
    <a href="{% url 'users:register' %}">
      register
    </a> -
    <a href="{% url 'users:login' %}">
      log in
    </a>
  {% endif %}
</p>

{% block content %}{% endblock content %}
```

The logout view
The logout_view() *function uses Django's* logout() *function and then redirects the user back to the home page. Since there is no logout page, there is no logout template. Make sure to write this code in the views.py file that's stored in the users app folder.*

```
from django.http import HttpResponseRedirect
from django.core.urlresolvers import reverse
from django.contrib.auth import logout

def logout_view(request):
    """Log the user out."""
    logout(request)
    return HttpResponseRedirect(
        reverse('learning_logs:index'))
```

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse
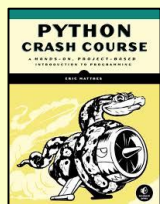
User accounts (cont.)

The register view
The register view needs to display a blank registration form when the page is first requested, and then process completed registration forms. A successful registration logs the user in and redirects to the home page.

```
from django.contrib.auth import login
from django.contrib.auth import authenticate
from django.contrib.auth.forms import \
        UserCreationForm

def register(request):
    """Register a new user."""
    if request.method != 'POST':
        # Show blank registration form.
        form = UserCreationForm()
    else:
        # Process completed form.
        form = UserCreationForm(
                data=request.POST)

        if form.is_valid():
            new_user = form.save()
            # Log in, redirect to home page.
            pw = request.POST['password1']
            authenticated_user = authenticate(
                username=new_user.username,
                password=pw
            )
            login(request, authenticated_user)
            return HttpResponseRedirect(
                reverse('learning_logs:index'))

    context = {'form': form}
    return render(request,
        'users/register.html', context)
```

Styling your project
The django-bootstrap3 app allows you to use the Bootstrap library to make your project look visually appealing. The app provides tags that you can use in your templates to style individual elements on a page. Learn more at http://django-bootstrap3.readthedocs.io/.

Deploying your project
Heroku lets you push your project to a live server, making it available to anyone with an internet connection. Heroku offers a free service level, which lets you learn the deployment process without any commitment. You'll need to install a set of heroku tools, and use git to track the state of your project. See http://devcenter.heroku.com/, and click on the Python link.

User accounts (cont.)

The register template
The register template displays the registration form in paragraph formats.

```
{% extends 'learning_logs/base.html' %}

{% block content %}

  <form method='post'
      action="{% url 'users:register' %}">

    {% csrf_token %}
    {{ form.as_p }}

    <button name='submit'>register</button>
    <input type='hidden' name='next'
      value="{% url 'learning_logs:index' %}"/>

  </form>

{% endblock content %}
```

Connecting data to users
Users will have data that belongs to them. Any model that should be connected directly to a user needs a field connecting instances of the model to a specific user.

Making a topic belong to a user
Only the highest-level data in a hierarchy needs to be directly connected to a user. To do this import the User model, and add it as a foreign key on the data model.
  After modifying the model you'll need to migrate the database. You'll need to choose a user ID to connect each existing instance to.

```
from django.db import models
from django.contrib.auth.models import User

class Topic(models.Model):
    """A topic the user is learning about."""
    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(
            auto_now_add=True)
    owner = models.ForeignKey(User)

    def __str__(self):
        return self.text
```

Querying data for the current user
In a view, the request object has a user attribute. You can use this attribute to query for the user's data. The filter() function then pulls the data that belongs to the current user.

```
topics = Topic.objects.filter(
                owner=request.user)
```

Connecting data to users (cont.)

Restricting access to logged-in users
Some pages are only relevant to registered users. The views for these pages can be protected by the @login_required decorator. Any view with this decorator will automatically redirect non-logged in users to an appropriate page. Here's an example views.py file.

```
from django.contrib.auth.decorators import /
        login_required
--snip--

@login_required
def topic(request, topic_id):
    """Show a topic and all its entries."""
```

Setting the redirect URL
The @login_required decorator sends unauthorized users to the login page. Add the following line to your project's settings.py file so Django will know how to find your login page.

```
LOGIN_URL = '/users/login/'
```

Preventing inadvertent access
Some pages serve data based on a parameter in the URL. You can check that the current user owns the requested data, and return a 404 error if they don't. Here's an example view.

```
from django.http import Http404

--snip--
def topic(request, topic_id):
    """Show a topic and all its entries."""
    topic = Topics.objects.get(id=topic_id)
    if topic.owner != request.user:
        raise Http404
    --snip--
```

Using a form to edit data
If you provide some initial data, Django generates a form with the user's existing data. Users can then modify and save their data.

Creating a form with initial data
The instance parameter allows you to specify initial data for a form.

```
form = EntryForm(instance=entry)
```

Modifying data before saving
The argument commit=False allows you to make changes before writing data to the database.

```
new_topic = form.save(commit=False)
new_topic.owner = request.user
new_topic.save()
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Acknowledgements & References

The exercises of this document are collected from different sources (e.g.,https://pynative.com, https://aprendeconalf.es/docencia/python, https://python.org, https://www.codesdope.com) and own material.

Some references to keep in mind:

- https://www.pythoncheatsheet.org

- https://towardsdatascience.com/collecting-data-science-cheat-sheets-d2cdff092855