Python Cheat Sheet - Keywords

"A puzzle a day to learn, code, and play" → Visit finxter.com

Keyword	Description	Code example
`False`, `True`	Data values from the data type Boolean	`False == (1 > 2), True == (2 > 1)`

`and`, `or`, `not`	Logical operators:
	`(x and y)` → both x and y must be True
	`(x or y)` → either x or y must be True
	`(not x)` → x must be false
		`x, y = True, False`
		`(x or y) == True       # True`
		`(x and y) == False     # True`
		`(not y) == True        # True`

`break`	Ends loop prematurely	`while(True):`
		`   break # no infinite loop`
		`print("hello world")`

`continue`	Finishes current loop iteration	`while(True):`
		`  continue`
		`  print("43") # dead code`

`class`
 	
`def`
	Defines a new class → a real-world concept (object oriented programming)
	
	Defines a new function or class method. For latter, first parameter ("self") points to the class object. When calling class method, first parameter is implicit.
		`class Beer:`
		`   def __init__(self):`
		`       self.content = 1.0`
		`   def drink(self):`
		`       self.content = 0.0`
		
		`becks = Beer() # constructor - create class`
		`becks.drink() # beer empty: b.content == 0`

`if`, `elif`, `else`	Conditional program execution: program starts with "if" branch, tries the "elif" branches, and finishes with "else" branch (until one branch evaluates to True).	`x = int(input("your value: "))`
		`if x > 3: print("Big")`
		`elif x == 3: print("Medium")`
		`else: print("Small")`

`for`, `while`	`# For loop declaration`
	`for i in [0,1,2]:`
	`   print(i)`
		`# While loop - same semantics`
		`j = 0`
		`while j < 3:`
		`   print(j)`
		`   j = j + 1`

`in`	Checks whether element is in sequence	`42 in [2, 39, 42] # True`

`is`	Checks whether both elements point to the same object	`y = x = 3`
		`x is y # True`
		`[3] is [3] # False`

`None`	Empty value constant	`def f():`
		`   x = 2`
		`f() is None # True`

`lambda`	Function with no name (anonymous function)	`(lambda x: x + 3)(3) # returns 6`

`return`	Terminates execution of the function and passes the flow of execution to the caller. An optional value after the return keyword specifies the function result.	`def incrementor(x):`
		`   return x + 1`
		`incrementor(4) # returns 5`

finxter

Python Cheat Sheet - Basic Data Types

“A puzzle a day to learn, code, and play” ➜ Visit finxter.com

Boolean

Description

The Boolean data type is a truth value, either `True` or `False`.

The Boolean operators ordered by priority:

`not x` ➜ “if x is False, then x, else y”
`x and y` ➜ “if x is False, then x, else y”
`x or y` ➜ “if x is False, then y, else x”

These comparison operators evaluate to `True`:

```
1 < 2 and 0 <= 1 and 3 > 2 and 2 >=2 and
1 == 1 and 1 != 0 # True
```

Example

```
## 1. Boolean Operations
x, y = True, False
print(x and not y) # True
print(not x and y or x) # True

## 2. If condition evaluates to False
if None or 0 or 0.0 or '' or [] or {} or set():
    # None, 0, 0.0, empty strings, or empty
    # container types are evaluated to False
    print("Dead code") # Not reached
```

Integer, Float

Description

An integer is a positive or negative number without floating point (e.g. `3`). A float is a positive or negative number with floating point precision (e.g. `3.14159265359`).

The ‘`//`’ operator performs integer division. The result is an integer value that is rounded towards the smaller integer number (e.g. `3 // 2 == 1`).

Example

```
## 3. Arithmetic Operations
x, y = 3, 2
print(x + y) # = 5
print(x - y) # = 1
print(x * y) # = 6
print(x / y) # = 1.5
print(x // y) # = 1
print(x % y) # = 1s
print(-x) # = -3
print(abs(-x)) # = 3
print(int(3.9)) # = 3
print(float(3)) # = 3.0
print(x ** y) # = 9
```

String

Description

Python Strings are sequences of characters.

The four main ways to create strings are the following.

1. Single quotes
`'Yes'`
2. Double quotes
`"Yes"`
3. Triple quotes (multi-line)

```
"""Yes
We Can"""
```

4. String method
`str(5) == '5' # True`
5. Concatenation
`"Ma" + "hatma" # 'Mahatma'`

These are whitespace characters in strings.

- Newline `\n`
- Space `\s`
- Tab `\t`

Example

```
## 4. Indexing and Slicing
s = "The youngest pope was 11 years old"
print(s[0])       # 'T'
print(s[1:3])     # 'he'
print(s[-3:-1])   # 'ol'
print(s[-3:])     # 'old'
x = s.split()     # creates string array of words
print(x[-3] + " " + x[-1] + " " + x[2] + "s")
                  # '11 old popes'

## 5. Most Important String Methods
y = "    This is lazy\t\n   "
print(y.strip()) # Remove Whitespace: 'This is lazy'
print("DrDre".lower()) # Lowercase: 'drdre'
print("attention".upper()) # Uppercase: 'ATTENTION'
print("smartphone".startswith("smart")) # True
print("smartphone".endswith("phone")) # True
print("another".find("other")) # Match index: 2
print("cheat".replace("ch", "m")) # 'meat'
print(','.join(["F", "B", "I"])) # 'F,B,I'
print(len("Rumpelstiltskin")) # String length: 15
print("ear" in "earth") # Contains: True
```

finxter

Python Cheat Sheet - Complex Data Types

"A puzzle a day to learn, code, and play" → Visit finxter.com

	Description	Example

List	A container data type that stores a sequence of elements. Unlike strings, lists are mutable: modification possible.	`l = [1, 2, 2]`
		`print(len(l)) # 3`

Adding elements	Add elements to a list with (i) append, (ii) insert, or (iii) list concatenation.
	The append operation is very fast.
		`[1, 2, 2].append(4) # [1, 2, 2, 4]`
		`[1, 2, 4].insert(2,2) # [1, 2, 2, 4]`
		`[1, 2, 2] + [4] # [1, 2, 2, 4]`

Removal	Removing an element can be slower.	`[1, 2, 2, 4].remove(1) # [2, 2, 4]`
Reversing	This reverses the order of list elements.	`[1, 2, 3].reverse() # [3, 2, 1]`
Sorting	Sorts a list. The computational complexity of sorting is O(n log n) for n list elements.	`[2, 4, 2].sort() # [2, 2, 4]`

Indexing	Finds the first occurence of an element in the list & returns its index. Can be slow as the whole list is traversed.	`[2, 2, 4].index(2) # index of element 4 is "0"`
		`[2, 2, 4].index(2,1) # index of element 2 after pos 1 is "1"`

Stack	Python lists can be used intuitively as stack via the two list operations append() and pop().	`stack = [3]`
		`stack.append(42) # [3, 42]`
		`stack.pop() # 42 (stack: [3])`
		`stack.pop() # 3 (stack: [])`

Set	A set is an unordered collection of elements. Each can exist only once.	`basket = {'apple', 'eggs', 'banana', 'orange'}`
		`same = set(['apple', 'eggs', 'banana', 'orange'])`

Dictionary	The dictionary is a useful data structure for storing (key, value) pairs.	`calories = {'apple' : 52, 'banana' : 89, 'choco' : 546}`

Reading and writing elements	Read and write elements by specifying the key within the brackets. Use the keys() and values() functions to access all keys and values of the dictionary.	`print(calories['apple'] < calories['choco']) # True`
		`calories['cappu'] = 74`
		`print(calories['banana'] < calories['cappu']) # False`
		`print('apple' in calories.keys()) # True`
		`print(52 in calories.values()) # True`

Dictionary Looping	You can loop over the (key, value) pairs of a dictionary with the items() method.	`for k, v in calories.items():`
		`       print(k) if v > 500 else None # 'chocolate'`

Membership operator	Check with the 'in' keyword whether the set, list, or dictionary contains an element. Set containment is faster than list containment.	`basket = {'apple', 'eggs', 'banana', 'orange'}`
		`print('eggs' in basket} # True`
		`print('mushroom' in basket} # False`

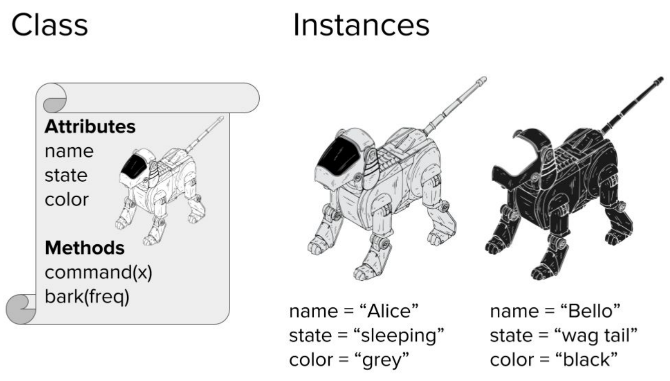
List and Set Comprehension	List comprehension is the concise Python way to create lists. Use brackets plus an expression, followed by a for clause. Close with zero or more for or if clauses.
	
	Set comprehension is similar to list comprehension.
		`# List comprehension`
		`l = [('Hi ' + x) for x in ['Alice', 'Bob', 'Pete']]`
		`print(l) # ['Hi Alice', 'Hi Bob', 'Hi Pete']`
		`l2 = [x * y for x in range(3) for y in range(3) if x>y]`
		`print(l2) # [0, 0, 2]`
		`# Set comprehension`
		`squares = { x**2 for x in [0,2,4] if x < 4 } # {0, 4}`

finxter

Python Cheat Sheet - Classes

“A puzzle a day to learn, code, and play” ➜ Visit finxter.com

	Description

Classes	A class encapsulates data and functionality - data as attributes, and functionality as methods. It is a blueprint to create concrete instances in the memory.
	 Class
	Instances
	Attributes
	name
	state
	color
	Methods
	command(x)
	bark(freq)
	name = “Alice”
	state = “sleeping”
	color = “grey”
	name = “Bello”
	state = “wag tail”
	color = “black”

Instance	You are an instance of the class human. An instance is a concrete implementation of a class: all attributes of an instance have a fixed value. Your hair is blond, brown, or black - but never unspecified. Each instance has its own attributes independent of other instances. Yet, class variables are different. These are data values associated with the class, not the instances. Hence, all instance share the same class variable `species` in the example.
Self	The first argument when defining any method is always the `self` argument. This argument specifies the instance on which you call the method. `self` gives the Python interpreter the information about the concrete instance. To *define* a method, you use `self` to modify the instance attributes. But to *call* an instance method, you do not need to specify `self`.

Creation	You can create classes “on the fly” and use them as logical units to store complex data types.
	`class Employee():`
	`        pass`
	`employee = Employee()`
	`employee.salary = 122000`
	`employee.firstname = "alice"`
	`employee.lastname = "wonderland"`
	
	`print(employee.firstname + " "`
	`        + employee.lastname + " "`
	`        + str(employee.salary) + "$")`
	`# alice wonderland 122000$`

Example

```python
class Dog:
        """ Blueprint of a dog """

        # class variable shared by all instances
        species = ["canis lupus"]

        def __init__(self, name, color):
                self.name = name
                self.state = "sleeping"
                self.color = color

        def command(self, x):
                if x == self.name:
                        self.bark(2)
                elif x == "sit":
                self.state = "sit"
                else:
                        self.state = "wag tail"

        def bark(self, freq):
                for i in range(freq):
                        print("[" + self.name
                              + "]: Woof!")


bello = Dog("bello", "black")
alice = Dog("alice", "white")

print(bello.color) # black
print(alice.color) # white

bello.bark(1) # [bello]: Woof!

alice.command("sit")
print("[alice]: " + alice.state)
# [alice]: sit

bello.command("no")
print("[bello]: " + bello.state)
# [bello]: wag tail

alice.command("alice")
# [alice]: Woof!
# [alice]: Woof!

bello.species += ["wulf"]
print(len(bello.species)
        == len(alice.species)) # True (!)
```

Python Cheat Sheet - Functions and Tricks

"A puzzle a day to learn, code, and play" → Visit finxter.com

		Description	Example	Result
ADVANCED FUNCTIONS	`map(func, iter)`	Executes the function on all elements of the iterable	`list(map(lambda x: x[0], ['red', 'green', 'blue']))`	`['r', 'g', 'b']`
	`map(func, i1, ..., ik)`	Executes the function on all k elements of the k iterables	`list(map(lambda x, y: str(x) + ' ' + y + 's' , [0, 2, 2], ['apple', 'orange', 'banana']))`	`['0 apples', '2 oranges', '2 bananas']`
	`string.join(iter)`	Concatenates iterable elements separated by `string`	`' marries '.join(list(['Alice', 'Bob']))`	`'Alice marries Bob'`
	`filter(func, iterable)`	Filters out elements in iterable for which function returns False (or 0)	`list(filter(lambda x: True if x>17 else False, [1, 15, 17, 18]))`	`[18]`
	`string.strip()`	Removes leading and trailing whitespaces of string	`print("\n   \t  42  \t ".strip())`	`42`
	`sorted(iter)`	Sorts iterable in ascending order	`sorted([8, 3, 2, 42, 5])`	`[2, 3, 5, 8, 42]`
	`sorted(iter, key=key)`	Sorts according to the key function in ascending order	`sorted([8, 3, 2, 42, 5], key=lambda x: 0 if x==42 else x)`	`[42, 2, 3, 5, 8]`
	`help(func)`	Returns documentation of func	`help(str.upper())`	`'... to uppercase.'`
	`zip(i1, i2, ...)`	Groups the i-th elements of iterators i1, i2, ... together	`list(zip(['Alice', 'Anna'], ['Bob', 'Jon', 'Frank']))`	`[('Alice', 'Bob'), ('Anna', 'Jon')]`
	Unzip	Equal to: 1) unpack the zipped list, 2) zip the result	`list(zip(*[('Alice', 'Bob'), ('Anna', 'Jon')]`	`[('Alice', 'Anna'), ('Bob', 'Jon')]`
	`enumerate(iter)`	Assigns a counter value to each element of the iterable	`list(enumerate(['Alice', 'Bob', 'Jon']))`	`[(0, 'Alice'), (1, 'Bob'), (2, 'Jon')]`
TRICKS	python -m http.server <P>	Share files between PC and phone? Run command in PC's shell. <P> is any port number 0–65535. Type < IP address of PC>:<P> in the phone's browser. You can now browse the files in the PC directory.		
	Read comic	`import antigravity`	Open the comic series xkcd in your web browser	
	Zen of Python	`import this`	`'...Beautiful is better than ugly. Explicit is ...'`	

	Swapping numbers	Swapping variables is a breeze in Python. No offense, Java!	`a, b = 'Jane', 'Alice'`
			`a, b = b, a`
				`a = 'Alice'`
				`b = 'Jane'`

	Unpacking arguments	Use a sequence as function arguments via asterisk operator *. Use a dictionary (key, value) via double asterisk operator **	`def f(x, y, z): return x + y * z`
			`f(*[1, 3, 4])`
			`f(**{'z' : 4, 'x' : 1, 'y' : 3})`
				
				`13`
				`13`

	Extended Unpacking	Use unpacking for multiple assignment feature in Python	`a, *b = [1, 2, 3, 4, 5]`	`a = 1`
				`b = [2, 3, 4, 5]`

	Merge two dictionaries	Use unpacking to merge two dictionaries into a single one	`x={'Alice' : 18}`
			`y={'Bob' : 27, 'Ann' : 22}`
			`z = {**x,**y}`
				`z = {'Alice': 18, 'Bob': 27, 'Ann': 22}`

finxter

Python Cheat Sheet: 14 Interview Questions

"A puzzle a day to learn, code, and play" ➡

FREE Python Email Course @ http://bit.ly/free-python-course

Question | **Code**

Check if list contains integer x

```python
l = [3, 3, 4, 5, 2, 111, 5]
print(111 in l) # True
```

Get missing number in [1...100]

```python
def get_missing_number(lst):
   return set(range(lst[len(lst)-1])[1:]) - set(l)
l = list(range(1,100))
l.remove(50)
print(get_missing_number(l)) # 50
```

Find duplicate number in integer list

```python
def find_duplicates(elements):
   duplicates, seen = set(), set()
   for element in elements:
      if element in seen:
         duplicates.add(element)
      seen.add(element)
   return list(duplicates)
```

Compute the intersection of two lists

```python
def intersect(lst1, lst2):
   res, lst2_copy = [], lst2[:]
   for el in lst1:
      if el in lst2_copy:
          res.append(el)
          lst2_copy.remove(el)
   return res
```

Check if two strings are anagrams

```python
def is_anagram(s1, s2):
   return set(s1) == set(s2)
print(is_anagram("elvis", "lives")) # True
```

Find max and min in unsorted list

```python
l = [4, 3, 6, 3, 4, 888, 1, -11, 22, 3]
print(max(l)) # 888
print(min(l)) # -11
```

Remove all duplicates from list

```python
lst = list(range(10)) + list(range(10))
lst = list(set(lst))
print(lst)
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Reverse string using recursion

```python
def reverse(string):
    if len(string)<=1: return string
    return reverse(string[1:])+string[0]
print(reverse("hello")) # olleh
```

Find pairs of integers in list so that their sum is equal to integer x

```python
def find_pairs(l, x):
   pairs = []
   for (i, el_1) in enumerate(l):
      for (j, el_2) in enumerate(l[i+1:]):
         if el_1 + el_2 == x:
            pairs.append((el_1, el_2))
   return pairs
```

Compute the first n Fibonacci numbers

```python
a, b = 0, 1
n = 10
for i in range(n):
    print(b)
    a, b = b, a+b
# 1, 1, 2, 3, 5, 8, ...
```

Check if a string is a palindrome

```python
def is_palindrome(phrase):
    return phrase == phrase[::-1]
print(is_palindrome("anna")) # True
```

Sort list with Quicksort algorithm

```python
def qsort(L):
    if L == []: return []
    return qsort([x for x in L[1:] if x< L[0]]) + L[0:1] +
qsort([x for x in L[1:] if x>=L[0]])
lst = [44, 33, 22, 5, 77, 55, 999]
print(qsort(lst))
# [5, 22, 33, 44, 55, 77, 999]
```

Use list as stack, array, and queue

```python
# as a list ...
l = [3, 4]
l +=  [5, 6] # l = [3, 4, 5, 6]

# ... as a stack ...
l.append(10) # l = [4, 5, 6, 10]
l.pop() # l = [4, 5, 6]

# ... and as a queue
l.insert(0, 5) # l = [5, 4, 5, 6]
l.pop() # l = [5, 4, 5]
```

Find all permutations of string

```python
def get_permutations(w):
    if len(w)<=1:
        return set(w)
    smaller = get_permutations(w[1:])
    perms = set()
    for x in smaller:
        for pos in range(0,len(x)+1):
            perm = x[:pos] + w[0] + x[pos:]
            perms.add(perm)
    return perms
print(get_permutations("nan"))
# {'nna', 'ann', 'nan'}
```

finxter

Python Cheat Sheet: NumPy

"A puzzle a day to learn, code, and play" ➔ Visit finxter.com

Name	Description	Example

`a.shape`	The shape attribute of NumPy array a keeps a tuple of integers. Each integer describes the number of elements of the axis.	`a = np.array([[1,2],[1,1],[0,0]])`
		`print(np.shape(a))        # (3, 2)`

`a.ndim`	The ndim attribute is equal to the length of the shape tuple.	`print(np.ndim(a))          # 2`

*	The asterisk (star) operator performs the Hadamard product, i.e., multiplies two matrices with equal shape element-wise.	`a = np.array([[2, 0], [0, 2]])`
		`b = np.array([[1, 1], [1, 1]])`
		`print(a*b)          # [[2 0] [0 2]]`

`np.matmul(a,b), a@b`	The standard matrix multiplication operator. Equivalent to the @ operator.	`print(np.matmul(a,b))`
		`# [[2 2] [2 2]]`

`np.arange([start, ]stop, [step, ])`	Creates a new 1D numpy array with evenly spaced values	`print(np.arange(0,10,2))`
		`# [0 2 4 6 8]`

`np.linspace(start, stop, num=50)`	Creates a new 1D numpy array with evenly spread elements within the given interval	`print(np.linspace(0,10,3))`
		`# [ 0.  5. 10.]`

`np.average(a)`	Averages over all the values in the numpy array	`a = np.array([[2, 0], [0, 2]])`
		`print(np.average(a))          # 1.0`

`<slice> = <val>`	Replace the <slice> as selected by the slicing operator with the value <val>.	`a = np.array([0, 1, 0, 0, 0])`
		`a[::2] = 2`
		`print(a)             # [2 1 2 0 2]`

`np.var(a)`	Calculates the variance of a numpy array.	`a = np.array([2, 6])`
		`print(np.var(a))              # 4.0`

`np.std(a)`	Calculates the standard deviation of a numpy array	`print(np.std(a))              # 2.0`

`np.diff(a)`	Calculates the difference between subsequent values in NumPy array a	`fibs = np.array([0, 1, 1, 2, 3, 5])`
		`print(np.diff(fibs, n=1))`
		`# [1 0 1 1 2]`

`np.cumsum(a)`	Calculates the cumulative sum of the elements in NumPy array a.	`print(np.cumsum(np.arange(5)))`
		`# [ 0  1  3  6 10]`

`np.sort(a)`	Creates a new NumPy array with the values from a (ascending).	`a = np.array([10,3,7,1,0])`
		`print(np.sort(a))`
		`# [ 0  1  3  7 10]`

`np.argsort(a)`	Returns the indices of a NumPy array so that the indexed values would be sorted.	`a = np.array([10,3,7,1,0])`
		`print(np.argsort(a))`
		`# [4 3 1 2 0]`

`np.max(a)`	Returns the maximal value of NumPy array a.	`a = np.array([10,3,7,1,0])`
		`print(np.max(a))              # 10`

`np.argmax(a)`	Returns the index of the element with maximal value in the NumPy array a.	`a = np.array([10,3,7,1,0])`
		`print(np.argmax(a))           # 0`

`np.nonzero(a)`	Returns the indices of the nonzero elements in NumPy array a.	`a = np.array([10,3,7,1,0])`
		`print(np.nonzero(a))      # [0 1 2 3]`

finxter

Python Cheat Sheet: Object Orientation Terms

"A puzzle a day to learn, code, and play" → Visit finxter.com

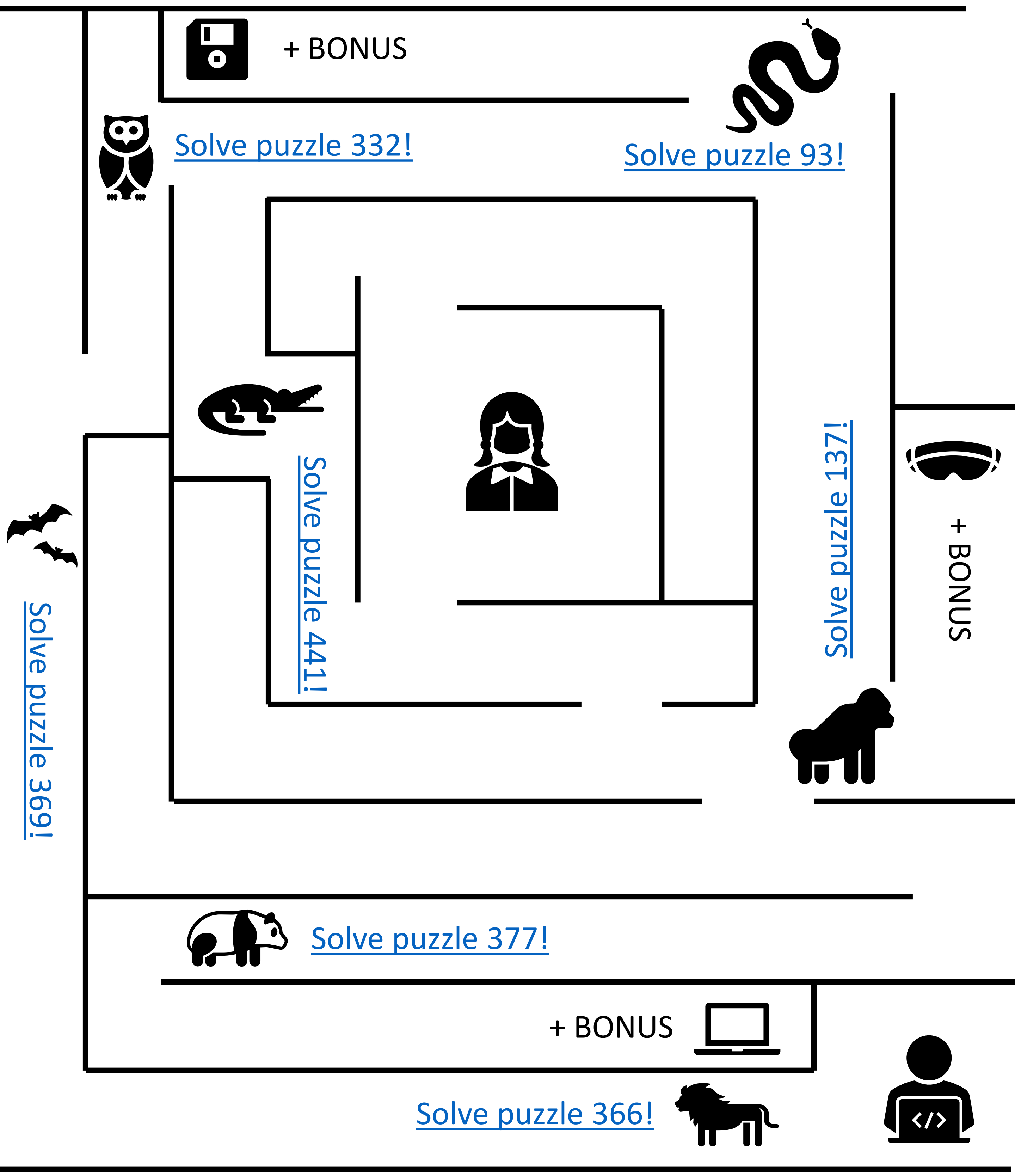
	Description
Class	A blueprint to create objects. It defines the data (attributes) and functionality (methods) of the objects. You can access both attributes and methods via the dot notation.
Object (=instance)	A piece of encapsulated data with functionality in your Python program that is built according to a class definition. Often, an object corresponds to a thing in the real world. An example is the object "Obama" that is created according to the class definition "Person". An object consists of an arbitrary number of attributes and methods, encapsulated within a single unit.
Instantiation	The process of creating an object of a class. This is done with the constructor method __init__(self, ...).
Method	A subset of the overall functionality of an object. The method is defined similarly to a function (using the keyword "def") in the class definition. An object can have an arbitrary number of methods.
Self	The first argument when defining any method is always the `self` argument. This argument specifies the instance on which you call the method. `self` gives the Python interpreter the information about the concrete instance. To *define* a method, you use `self` to modify the instance attributes. But to *call* an instance method, you do not need to specify `self`.
Encapsulation	Binding together data and functionality that manipulates the data.
Attribute	A variable defined for a class (class attribute) or for an object (instance attribute). You use attributes to package data into enclosed units (class or instance).
Class attribute	(=class variable, static variable, static attribute) A variable that is created statically in the class definition and that is shared by all class objects.
Instance attribute (=instance variable)	A variable that holds data that belongs only to a single instance. Other instances do not share this variable (in contrast to class attributes). In most cases, you create an instance attribute x in the constructor when creating the instance itself using the self keywords (e.g. self.x = <val>).
Dynamic attribute	An instance attribute that is defined dynamically during the execution of the program and that is not defined within any method. For example, you can simply add a new attribute `neew` to any object o by calling `o.neew = <val>`.
Method overloading	You may want to define a method in a way so that there are multiple options to call it. For example for class X, you define a method f(...) that can be called in three ways: f(a), f(a,b), or f(a,b,c). To this end, you can define the method with default parameters (e.g. f(a, b=None, c=None).
Inheritance	Class A can inherit certain characteristics (like attributes or methods) from class B. For example, the class "Dog" may inherit the attribute "number_of_legs" from the class "Animal". In this case, you would define the inherited class "Dog" as follows: "class Dog(Animal): ..."

Example

```python
class Dog:

    # class attribute
    is_hairy = True

    # constructor
    def __init__(self, name):
        # instance attribute
        self.name = name

    # method
    def bark(self):
        print("Wuff")


bello = Dog("bello")
paris = Dog("paris")

print(bello.name)
"bello"

print(paris.name)
"paris"


class Cat:

    # method overloading
    def miau(self, times=1):
        print("miau " * times)

fifi = Cat()

fifi.miau()
"miau "

fifi.miau(5)
"miau miau miau miau miau "

# Dynamic attribute
fifi.likes = "mice"
print(fifi.likes)
"mice"

# Inheritance
class Persian_Cat(Cat):
    classification = "Persian"

mimi = Persian_Cat()
print(mimi.miau(3))
"miau miau miau "


print(mimi.classification)
```

finxter

[Test Sheet] Help Alice Find Her Coding Dad!

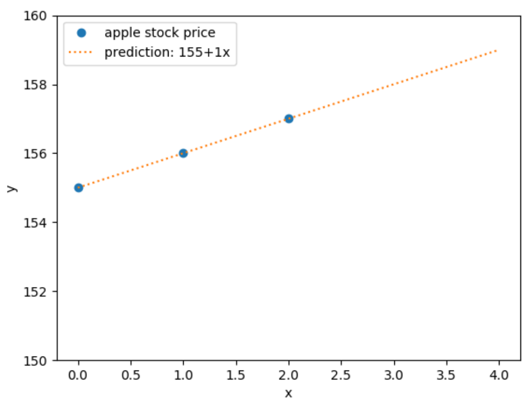


[Cheat Sheet] 6 Pillar Machine Learning Algorithms

Complete Course: https://academy.finxter.com/

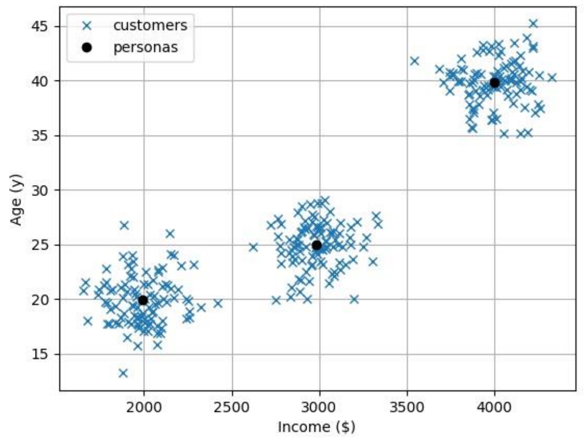
Linear Regression

https://blog.finxter.com/logistic-regression-in-one-line-python/



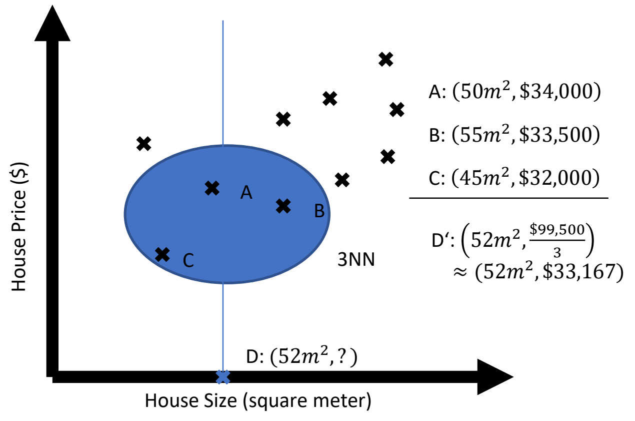
K-Means Clustering

https://blog.finxter.com/tutorial-how-to-run-k-means-clustering-in-1-line-of-python/



K Nearest Neighbors

https://blog.finxter.com/k-nearest-neighbors-as-a-python-one-liner/



A: $(50m^2, \$34{,}000)$

B: $(55m^2, \$33{,}500)$

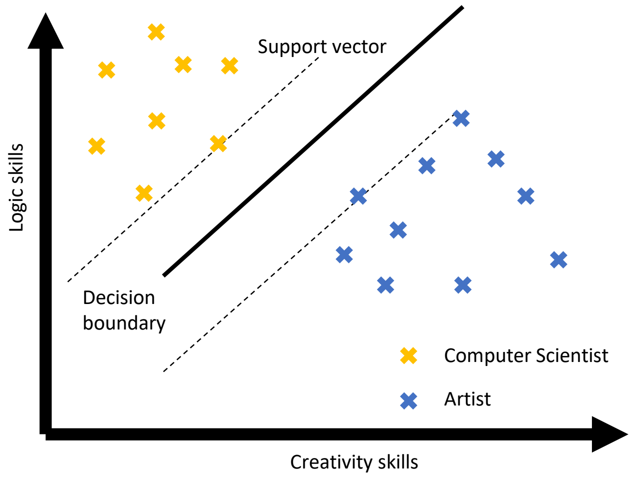
C: $(45m^2, \$32{,}000)$

D': $\left(52m^2, \frac{\$99{,}500}{3}\right)$
$\approx (52m^2, \$33{,}167)$

D: $(52m^2, ?)$

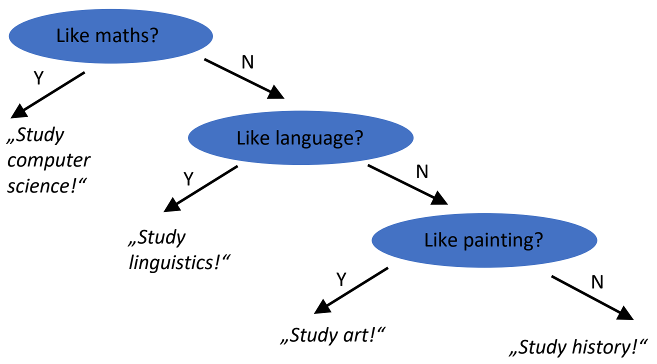
Support Vector Machine Classification

https://blog.finxter.com/support-vector-machines-python/



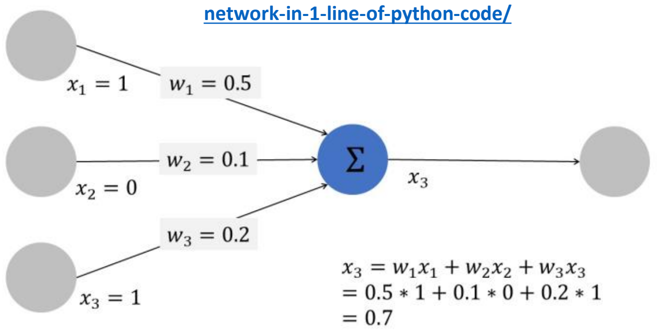
Decision Tree Classification

https://blog.finxter.com/decision-tree-learning-in-one-line-python/



Multilayer Perceptron

https://blog.finxter.com/tutorial-how-to-create-your-first-neural-network-in-1-line-of-python-code/



$x_3 = w_1x_1 + w_2x_2 + w_3x_3$
$= 0.5 * 1 + 0.1 * 0 + 0.2 * 1$
$= 0.7$

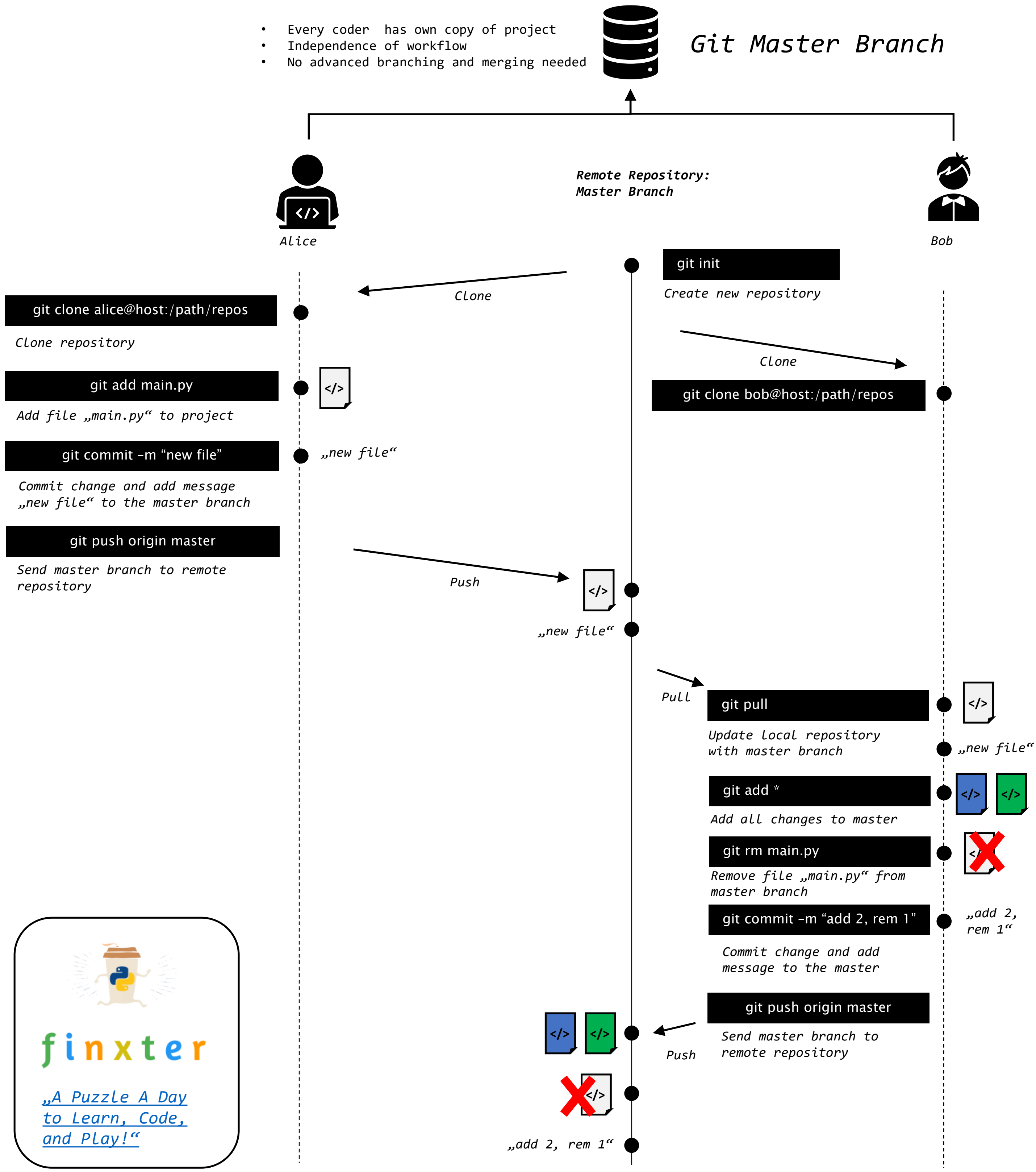
finxter

„Continuous Improvement in Your Coffee Break Python!“

The Simple Git Cheat Sheet – A Helpful Illustrated Guide

The Centralized Git Workflow

- Every coder has own copy of project
- Independence of workflow
- No advanced branching and merging needed



```
git init
```

Create new repository

```
git clone alice@host:/path/repos
```

Clone repository

```
git clone bob@host:/path/repos
```

```
git add main.py
```

Add file „main.py“ to project

```
git commit -m “new file”
```

Commit change and add message „new file“ to the master branch

```
git push origin master
```

Send master branch to remote repository

```
git pull
```

Update local repository with master branch

```
git add *
```

Add all changes to master

```
git rm main.py
```

Remove file „main.py“ from master branch

```
git commit –m “add 2, rem 1”
```

Commit change and add message to the master

```
git push origin master
```

Send master branch to remote repository

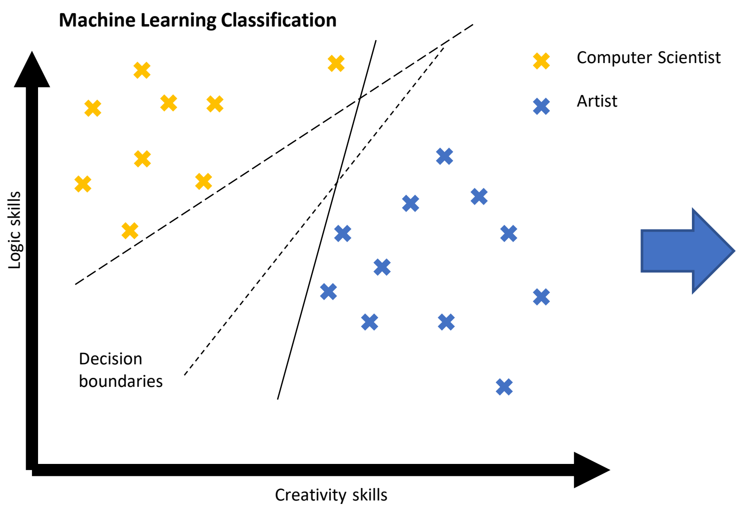
finxter

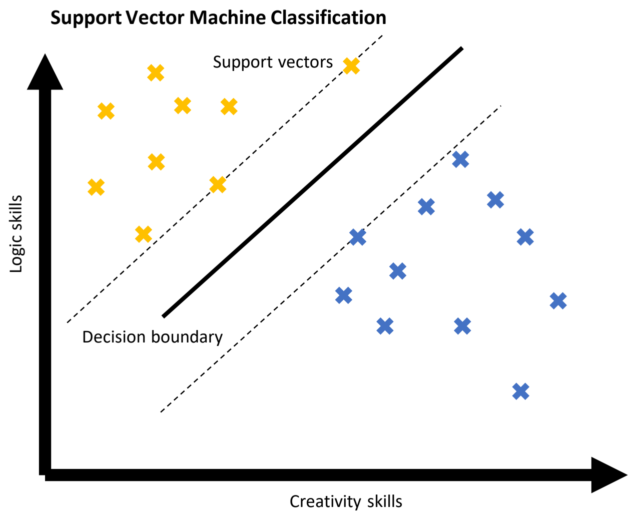
„A Puzzle A Day to Learn, Code, and Play!“

[Machine Learning Cheat Sheet] Support Vector Machines

Based on Article: https://blog.finxter.com/support-vector-machines-python/

Main idea: Maximize width of separator zone → increases „margin of safety" for classification





What are basic SVM properties?

Support Vector Machines	
Alternatives:	SVM, support-vector networks
Learning:	Classification, Regression
Advantages:	Robust for high-dimensional space
	Memory efficient (only uses support vectors)
	Flexible and customizable
Disadvantages:	Danger of overfitting in high-dimensional space
	No classification probabilities like Decision trees
Boundary:	Linear and Non-linear

What's the most basic Python code example?

```python
## Dependencies
from sklearn import svm
import numpy as np


## Data: student scores in (math, language, creativity)
## --> study field
X = np.array([[9, 5, 6, "computer science"],
              [10, 1, 2, "computer science"],
              [1, 8, 1, "literature"],
              [4, 9, 3, "literature"],
              [0, 1, 10, "art"],
              [5, 7, 9, "art"]])


## One-liner
svm = svm.SVC().fit(X[:,:-1], X[:,-1])


## Result & puzzle
student_0 = svm.predict([[3, 3, 6]])
print(student_0)

student_1 = svm.predict([[8, 1, 1]])
print(student_1)
```

What's the explanation of the code example?

Explanation: A Study Recommendation System with SVM

- NumPy array holds labeled training data (one row per user and one column per feature).
- Features: skill level in maths, language, and creativity.
- Labels: last column is recommended study field.
- 3D data → SVM separates data using 2D planes (the linear separator) rather than 1D lines.
- One-liner:
  1. Create model using constructor of scikit-learn's svm.SVC class (SVC = support vector classification).
  2. Call fit function to perform training based on labeled training data.
- Results: call predict function on new observations
  - student_0 (skills maths=3, language=3, and creativity=6) → SVM predicts "art"
  - student_1 (maths=8, language=1, and creativity=1) → SVM predicts "computer science"
- Final output of one-liner:

```python
## Result & puzzle
student_0 = svm.predict([[3, 3, 6]])
print(student_0)
# ['art']

student_1 = svm.predict([[8, 1, 1]])
print(student_1)
## ['computer science']
```

finxter

„A Puzzle A Day to Learn, Code, and Play!"

Python Cheat Sheet: List Methods

"A puzzle a day to learn, code, and play" ➜ Visit finxter.com

Method	Description	Example

`lst.append(x)`	Appends element `x` to the list `lst`.	`>>> l = []`
		`>>> l.append(42)`
		`>>> l.append(21)`
		`[42, 21]`

`lst.clear()`	Removes all elements from the list lst–which becomes empty.	`>>> lst = [1, 2, 3, 4, 5]`
		`>>> lst.clear()`
		`[]`

`lst.copy()`	Returns a copy of the list `lst`. Copies only the list, not the elements in the list (shallow copy).	`>>> lst = [1, 2, 3]`
		`>>> lst.copy()`
		`[1, 2, 3]`

`lst.count(x)`	Counts the number of occurrences of element `x` in the list `lst`.	`>>> lst = [1, 2, 42, 2, 1, 42, 42]`
		`>>> lst.count(42)`
		`3`
		`>>> lst.count(2)`
		`2`

`lst.extend(iter)`	Adds all elements of an iterable `iter` (e.g. another list) to the list `lst`.	`>>> lst = [1, 2, 3]`
		`>>> lst.extend([4, 5, 6])`
		`[1, 2, 3, 4, 5, 6]`

`lst.index(x)`	Returns the position (index) of the first occurrence of value `x` in the list `lst`.	`>>> lst = ["Alice", 42, "Bob", 99]`
		`>>> lst.index("Alice")`
		`0`
		`>>> lst.index(99, 1, 3)`
		`ValueError: 99 is not in list`

`lst.insert(i, x)`	Inserts element `x` at position (index) `i` in the list `lst`.	`>>> lst = [1, 2, 3, 4]`
		`>>> lst.insert(3, 99)`
		`[1, 2, 3, 99, 4]`

`lst.pop()`	Removes and returns the final element of the list `lst`.	`>>> lst = [1, 2, 3]`
		`>>> lst.pop()`
		`3`
		`>>> lst`
		`[1, 2]`

`lst.remove(x)`	Removes and returns the first occurrence of element `x` in the list `lst`.	`>>> lst = [1, 2, 99, 4, 99]`
		`>>> lst.remove(99)`
		`>>> lst`
		`[1, 2, 4, 99]`

`lst.reverse()`	Reverses the order of elements in the list `lst`.	`>>> lst = [1, 2, 3, 4]`
		`>>> lst.reverse()`
		`>>> lst`
		`[4, 3, 2, 1]`

`lst.sort()`	Sorts the elements in the list `lst` in ascending order.	`>>> lst = [88, 12, 42, 11, 2]`
		`>>> lst.sort()`
		`# [2, 11, 12, 42, 88]`
		`>>> lst.sort(key=lambda x: str(x)[0])`
		`# [11, 12, 2, 42, 88]`

The Ultimate Python Cheat Sheet

Keywords

Keyword	Description	Code Examples
`False`, `True`	Boolean data type	`False == (1 > 2)` `True == (2 > 1)` 👎 👍

`and`, `or`, `not`	Logical operators
	→ Both are true
	→ Either is true
	→ Flips Boolean
		`True and True    # True`
		`True or False    # True`
		`not False        # True`

`break`	Ends loop prematurely	`while True:`
		`   break # finite loop`

`continue`	Finishes current loop iteration	`while True:`
		`  continue`
		`  print("42") # dead code`

`class`	Defines new class	`class Coffee:`
		`   # Define your class`

`def`	Defines a new function or class method.	`def say_hi():`
		`   print('hi')`

`if`, `elif`, `else`	Conditional execution:
	- "if" condition == True?
	- "elif" condition == True?
	- Fallback: else branch
		`x = int(input("ur val:"))`
		`if   x > 3:  print("Big")`
		`elif x == 3: print("3")`
		`else:        print("Small")`

`for`, `while`	`# For loop`
	`for i in [0,1,2]:`
	`   print(i)`
		`# While loop does same`
		`j = 0`
		`while j < 3:`
		`   print(j); j = j + 1`

`in`	Sequence membership	`42 in [2, 39, 42] # True`

`is`	Same object memory location	`y = x = 3`
		`x is y       # True`
		`[3] is [3]   # False`

`None`	Empty value constant	`print() is None # True`
`lambda`	Anonymous function	`(lambda x: x+3)(3) # 6`

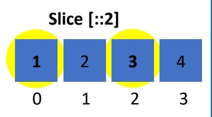
`return`	Terminates function. Optional return value defines function result.	`def increment(x):`
		`   return x + 1`
		`increment(4) # returns 5`

Basic Data Structures

Type	Description	Code Examples

Boolean	The Boolean data type is either `True` or `False`. Boolean operators are ordered by priority:
	`not` → `and` → `or`
	{ } → 👎
	{1, 2, 3} → 👍
		`## Evaluates to True:`
		`1<2 and 0<=1 and 3>2 and 2>=2 and 1==1 and 1!=0`
		
		`## Evaluates to False:`
		`bool(None or 0 or 0.0 or '' or [] or {} or set())`
		
		Rule: *None, 0, 0.0, empty strings, or empty container types evaluate to False*

Integer, Float	An **integer** is a positive or negative number without decimal point such as `3`.
	
	A **float** is a positive or negative number with floating point precision such as `3.1415926`.
	
	Integer division rounds toward the smaller integer (example: `3//2==1`).
		`## Arithmetic Operations`
		`x, y = 3, 2`
		`print(x + y)      # = 5`
		`print(x - y)      # = 1`
		`print(x * y)      # = 6`
		`print(x / y)      # = 1.5`
		`print(x // y)     # = 1`
		`print(x % y)      # = 1`
		`print(-x)         # = -3`
		`print(abs(-x))    # = 3`
		`print(int(3.9))   # = 3`
		`print(float(3))   # = 3.0`
		`print(x ** y)     # = 9`

String	Python Strings are sequences of characters.
	
	String Creation Methods:
	1. Single quotes
	`>>> 'Yes'`
	2. Double quotes
	`>>> "Yes"`
	3. Triple quotes (multi-line)
	`>>> """Yes`
	`    We Can"""`
	4. String method
	`>>> str(5) == '5'`
	`True`
	5. Concatenation
	`>>> "Ma" + "hatma"`
	`'Mahatma'`
	
	Whitespace chars:
	Newline \n,
	Space \s,
	Tab \t
		`## Indexing and Slicing`
		`s = "The youngest pope was 11 years"`
		`s[0] # 'T'`
		`s[1:3] # 'he'`
		`s[-3:-1] # 'ar'`
		`s[-3:] # 'ars'`
		
		`x = s.split()`
		`x[-2] + " " + x[2] + "s" # '11 popes'`
		
		`## String Methods`
		`y = "    Hello world\t\n   "`
		`y.strip() # Remove Whitespace`
		`"HI".lower() # Lowercase: 'hi'`
		`"hi".upper() # Uppercase: 'HI'`
		`"hello".startswith("he") # True`
		`"hello".endswith("lo") # True`
		`"hello".find("ll") # Match at 2`
		`"cheat".replace("ch", "m") # 'meat'`
		`''.join(["F", "B", "I"]) # 'FBI'`
		`len("hello world") # Length: 15`
		`"ear" in "earth" # True`

Complex Data Structures

Type	Description	Example
List	Stores a sequence of elements. Unlike strings, you can modify list objects (they're *mutable*).	`l = [1, 2, 2]` `print(len(l)) # 3` 🟦—🟨—🟩

Adding elements	Add elements to a list with (i) append, (ii) insert, or (iii) list concatenation.	`[1, 2].append(4) # [1, 2, 4]`
		`[1, 4].insert(1,9) # [1, 9, 4]`
		`[1, 2] + [4] # [1, 2, 4]`

Removal	Slow for lists	`[1, 2, 2, 4].remove(1) # [2, 2, 4]`
Reversing	Reverses list order	`[1, 2, 3].reverse() # [3, 2, 1]`
Sorting	Sorts list using fast Timsort	`[2, 4, 2].sort() # [2, 2, 4]`

Indexing	Finds the first occurrence of an element & returns index. Slow worst case for whole list traversal.	`[2, 2, 4].index(2)`
		`# index of item 2 is 0`
		`[2, 2, 4].index(2,1)`
		`# index of item 2 after pos 1 is 1`

Stack	Use Python lists via the list operations append() and pop()	`stack = [3]`
		`stack.append(42) # [3, 42]`
		`stack.pop() # 42 (stack: [3])`
		`stack.pop() # 3 (stack: [])`

Set	An unordered collection of unique elements (*at-most-once*) → fast membership *O(1)*	`basket = {'apple', 'eggs',`
		`          'banana', 'orange'}`
		`same = set(['apple', 'eggs',`
		`            'banana', 'orange'])`

Type	Description	Example

Dictionary	Useful data structure for storing (key, value) pairs	`cal = {'apple' : 52, 'banana' : 89,`
		`       'choco' : 546} # calories`

Reading and writing elements	Read and write elements by specifying the key within the brackets. Use the **`keys()`** and **`values()`** functions to access all keys and values of the dictionary	`print(cal['apple'] < cal['choco'])`
		`# True`
		`cal['cappu'] = 74`
		`print(cal['banana'] < cal['cappu'])`
		`# False`
		`print('apple' in cal.keys()) # True`
		`print(52 in cal.values())    # True`

Dictionary Iteration	You can access the (key, value) pairs of a dictionary with the **`items()`** method.	`for k, v in cal.items():`
		`    print(k) if v > 500 else ''`
		`# 'choco'`

Member-ship operator	Check with the **`in`** keyword if set, list, or dictionary contains an element. Set membership is faster than list membership.	`basket = {'apple', 'eggs',`
		`          'banana', 'orange'}`
		`print('eggs' in basket)      # True`
		`print('mushroom' in basket) # False`

List & set comprehe nsion	List comprehension is the concise Python way to create lists. Use brackets plus an expression, followed by a for clause. Close with zero or more for or if clauses.
	Set comprehension works similar to list comprehension.
		`l = ['hi ' + x for x in ['Alice',`
		`'Bob', 'Pete']]`
		`# ['Hi Alice', 'Hi Bob', 'Hi Pete']`
		
		`l2 = [x * y for x in range(3) for y`
		`in range(3) if x>y] # [0, 0, 2]`
		
		`squares = { x**2 for x in [0,2,4]`
		`if x < 4 } # {0, 4}`

*Subscribe to the **11x FREE** Python Cheat Sheet Course:*
https://blog.finxter.com/python-cheat-sheets/

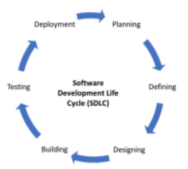
finxter Book: **Simplicity - The Finer Art of Creating Software**

Complexity

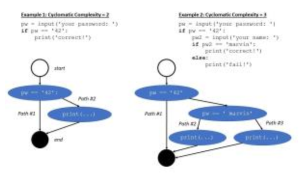
"A whole, made up of parts—difficult to analyze, understand, or explain".

Complexity appears in
- Project Lifecycle
- Code Development
- Algorithmic Theory
- Processes
- Social Networks
- Learning & Your Daily Life

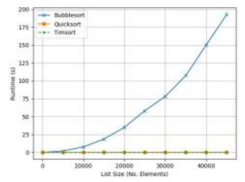
Project Lifecycle



Cyclomatic Complexity


Example 1: Cyclomatic Complexity = 2
Example 2: Cyclomatic Complexity = 3
start
end
Path #1
Path #2
Path #3

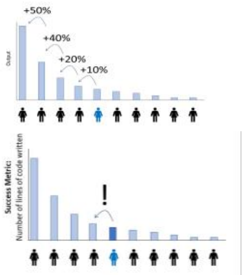
Runtime Complexity



→ Complexity reduces productivity and focus. It'll consume your precious time. Keep it simple!

80/20 Principle

Majority of effects come from the minority of causes.

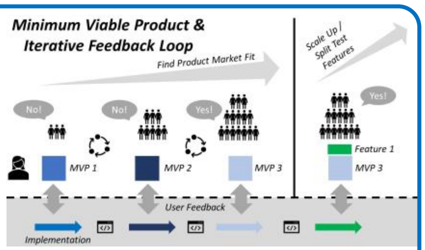


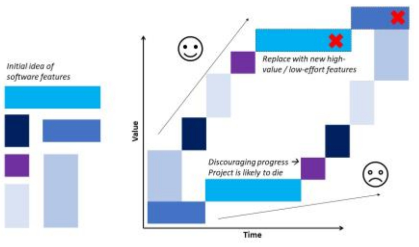
Pareto Tips
1. Figure out your success metrics.
2. Figure out your big goals in life.
3. Look for ways to achieve the same things with fewer resources.
4. Reflect on your own successes
5. Reflect on your own failures
6. Read more books in your industry.
7. Spend much of your time improving and tweaking existing products
8. Smile.
9. Don't do things that reduce value

Maximize Success Metric:
#lines of code written

Minimum Viable Product (MVP)

A minimum viable product in the software sense is code that is stripped from all features to focus on the core functionality.




Initial idea of software features
Value
Replace with new high-value / low-effort features
Discouraging progress → Project is likely to die
Time

How to MVP?
- Formulate hypothesis
- Omit needless features
- Split test to validate each new feature
- Focus on product-market fit
- Seek high-value and low-cost features

Clean Code Principles

1. You Ain't Going to Need It
2. The Principle of Least Surprise
3. Don't Repeat Yourself
4. **Code For People Not Machines**
5. Stand on the Shoulders of Giants
6. Use the Right Names
7. Single-Responsibility Principle
8. Use Comments
9. Avoid Unnecessary Comments
10. Be Consistent
11. Test
12. Think in Big Pictures
13. Only Talk to Your Friends
14. Refactor
15. Don't Overengineer
16. Don't Overuse Indentation
17. Small is Beautiful
18. Use Metrics
19. Boy Scout Rule: Leave Camp Cleaner Than You Found It

Unix Philosophy

1. Simple's Better Than Complex
2. **Small is Beautiful (*Again*)**
3. Make Each Program Do One Thing Well
4. Build a Prototype First
5. Portability Over Efficiency
6. Store Data in Flat Text Files
7. Use Software Leverage
8. Avoid Captive User Interfaces
9. **Program = Filter**
10. Worse is Better
11. Clean > Clever Code
12. **Design *Connected* Programs**
13. Make Your Code Robust
14. Repair What You Can — But Fail Early and Noisily
15. Write Programs to Write Programs

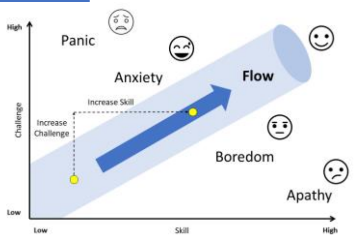
Premature Optimization

"Programmers waste enormous amounts of time thinking about […] the speed of noncritical parts of their programs. We should forget about small efficiencies, say about 97 % of the time: premature optimization is the root of all evil." – **Donald Knuth**

Performance Tuning 101
1. Measure, then improve
2. Focus on the slow 20%
3. Algorithmic optimization wins
4. All hail to the cache
5. Solve an easier problem version
6. Know when to stop

Flow

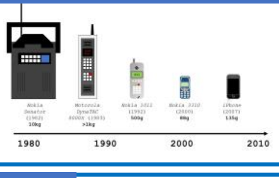
"… the source code of ultimate human performance" – ***Kotler***



How to Achieve Flow? (1) clear goals, (2) immediate feedback, and (3) balance opportunity & capacity.

Flow Tips for Coders
1. Always work on an explicit practical code project
2. Work on fun projects that fulfill your purpose
3. Perform from your strengths
4. Big chunks of coding time
5. Reduce distractions: smartphone + social
6. Sleep a lot, eat healthily, read quality books, and exercise → garbage in, garbage out!

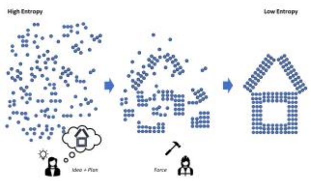
Less Is More in Design



How to Simplify Design?
1. Use whitespace
2. Remove design elements
3. Remove features
4. Reduce variation of fonts, font types, colors
5. Be consistent across UIs

Focus

You can take raw resources and move them from a state of high entropy into a state of low entropy—using *focused effort towards the attainment of a greater plan*.



3-Step Approach of Efficient Software Creation
1. Plan your code
2. Apply focused effort to make it real.
3. Seek feedback

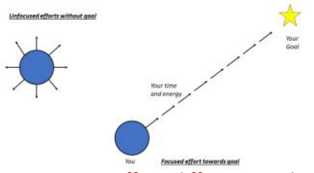


Figure: Same effort, different result.

*Subscribe to the **11x FREE** Python Cheat Sheet Course:*
https://blog.finxter.com/python-cheat-sheets/