Introduction to Python

Python Conceptual Hierarchy

1. Programs are composed of modules.
2. Modules contain statements.
3. Statements contain expressions.
4. *Expressions create and process objects.*

Data Types

Object type	Example literals/creation
Numbers	`1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()`
Strings	`'spam', "Bob's", b'a\x01c', u'sp\xc4m'`
Lists	`[1, [2, 'three'], 4.5], list(range(10))`
Dictionaries	`{'food': 'spam', 'taste': 'yum'}, dict(hours=10)`
Tuples	`(1, 'spam', 4, 'U'), tuple('spam'), namedtuple`
Files	`open('eggs.txt'), open(r'C:\ham.bin', 'wb')`
Sets	`set('abc'), {'a', 'b', 'c'}`
Other core types	Booleans, types, None

Operators

Arithmetic Operations

Operator	Name	Description
`a + b`	Addition	Sum of a and b
`a - b`	Subtraction	Difference of a and b
`a * b`	Multiplication	Product of a and b
`a / b`	True division	Quotient of a and b
`a // b`	Floor division	Quotient of a and b, removing fractional parts
`a % b`	Modulus	Remainder after division of a by b
`a ** b`	Exponentiation	a raised to the power of b
`-a`	Negation	The negative of a
`+a`	Unary plus	a unchanged (rarely used)

Comparison Operations

Operation	Description
`a == b`	a equal to b
`a != b`	a not equal to b
`a < b`	a less than b
`a > b`	a greater than b
`a <= b`	a less than or equal to b
`a >= b`	a greater than or equal to b

Bitwise Operations

Operator	Name	Description
`a & b`	Bitwise AND	Bits defined in both a and b
`a | b`	Bitwise OR	Bits defined in a or b or both
`a ^ b`	Bitwise XOR	Bits defined in a or b but not both
`a << b`	Bit shift left	Shift bits of a left by b units
`a >> b`	Bit shift right	Shift bits of a right by b units
`~a`	Bitwise NOT	Bitwise negation of a

Boolean Operators

Operation	Result
`x or y`	if *x* is false, then *y*, else *x*
`x and y`	if *x* is false, then *x*, else *y*
`not x`	if *x* is false, then `True`, else `False`

Identity and Membership Operators

Operator	Description
`a is b`	True if a and b are identical objects
`a is not b`	True if a and b are not identical objects
`a in b`	True if a is a member of b
`a not in b`	True if a is not a member of b

Expressions

- A combination of **objects** and **operators** that **computes a value** when executed
    - For instance:
        to add two numbers X and Y you would say X + Y,

- Rules:

Mixed operators follow operator precedence

Parentheses group subexpressions

Mixed types are converted up

Operator overloading and polymorphism

Data Types - Lists

- Ordered Collections of Arbitrary Objects
  - Lists are places to collect other objects so that you can treat them as groups.
  - Lists maintain left-to-right positional ordering among the items they contain (starting from index 0).
- Accessed by Offset
  - You can fetch an object from the list by indexing the list on the object's offset.
    i.e - object = list[index]
- Variable-length, Heterogeneous, and Arbitrarily Nestable
  - Lists can grow and shrink in place.
  - Can contain any type of objects including lists. We can have lists of lists.
- Of the Category "Mutable Sequence"
  - Lists support all the sequence operations we saw with strings.
  - These operations do not return a new list, but instead changes the object in place.
  - Lists also support the operations associated with mutable sequence objects.
- Arrays of Object References
  - Python lists are arrays not linked structures.
  - Reminiscent of a C array of pointers.

Data Types - Tuples

- Ordered collections of arbitrary objects
  - Like strings and lists, tuples are positionally ordered collections of objects.
  - Like lists, they can embed any kind of object.
- Accessed by offset
  - Like strings and lists, items in a tuple are accessed by offset
  - They support all the offset-based access operations, such as indexing and slicing.
- Of the category “immutable sequence”
  - Like strings and lists, tuples are sequences
  - However, like strings, tuples are immutable; they don’t support any of the in-place change operations applied to lists.
- Fixed-length, heterogeneous, and arbitrarily nestable
  - Because tuples are immutable, you cannot change the size of a tuple.
  - Tuples can hold any type of object.
- Arrays of object references
  - Like lists, tuples are best thought of as object reference arrays.

Data Types - Dictionary

- Accessed by key, not offset position
  - Dictionaries associate a set of values with keys.
  - You can fetch an item out of a dictionary using the key under which you originally stored it.
- Unordered collections of arbitrary objects
  - Unlike in a list, items stored in a dictionary aren't kept in any particular order.
- Variable-length, heterogeneous, and arbitrarily nestable
  - Like lists, dictionaries can grow and shrink in place.
  - They can contain objects of any type.
  - Each key can have just one associated value, but that value can be a collection type.
- Of the category "mutable mapping"
  - Dictionaries don't support the sequence operations that work on strings and lists.
- Tables of object references (hash tables)
  - Dictionaries are implemented as hash tables (data structures that support very fast retrieval), which start small and grow on demand.
  - Like lists, dictionaries store object references.

Statements

- **Expressions** process **objects** and are embedded in **statements**.
- **Statements** code the larger logic of a program's operation
  - they use and direct expressions to process the objects
- Statements are where objects spring into existence
  - e.g., in expressions within assignment statements
    X = Y + Z

If Statement

General Format:

```
if test1 :
        Body

elif test2 :
        Body

 else :
        Body
```

- It selects exactly one of the bodies by evaluating the tests one by one until one is found to be true.
- That body is executed (and no other body is executed).
- If all expressions are false, the body of the else clause, if present, is executed.

While Loop

```
while test:
        Body

else :
        Body
```

This repeatedly tests the test expression and, if it is true, executes the body.

If the expression is false the suite of the else clause, if present, is executed and the loop terminates.

A break statement executed in the first body terminates the loop without executing the else clause’s body.

A continue statement executed in the first body skips the rest of the body and goes back to testing the expression.

For Loop

```
for target in sequence/iterable object :
        Body
else :
        Body
```

The header line specifies an assignment target along with the object you want to step through.

When the for loop runs, it assigns the items in the iterable object to the target one by one and executes the loop body for each.

When the the end of the object is reached without encountering a break, the body in the else clause, if present, is executed, and the loop terminates.

A break statement executed terminates the loop without executing the else clause's body.

A continue statement executed in the first body skips the rest of the body and continues with the next item, or with the else clause if there is no next item.

Iteration

- An object in Python is considered **iterable** if it is either:
  - Physically stored in sequence
  - It is an object that produces one result at a time in the context of an iteration tool like a for loop
- An object in Python is an **iterator** if it follows the **iteration protocol**:
  - The object has a **__next__** method to advance to the next result.
  - The method raises **StopIteration** at the end of the series of results.

Iteration

Full Iteration Protocol

- Two objects: iterable object, iterator object
    - You initialize iteration for an iterable object, **iter(obj) runs objects __iter__().**
    - This returns an iterator object.
    - The iterator produces the values during iteration, **next(obj) runs objects __next__()**.
    - The iterator raises **StopIteration** when it has stepped through the entire sequence.
- Some Python objects only support one iterator operating on it at a time, such as files. In this case the first step can be forgone because these object are their own iterators.
- Other Python objects support multiple iterators, such as lists. For these objects we need to create iterators using **iter**.

Loop Coding Techniques

- The for loop is generally simpler to code and often quicker than a while loop.
- Resist the temptation to count things in Python - use iteration tools instead.
- Python provides a set of built-ins that allow you to specialize the iteration in a for loop.
  - Range function - produces a series of successively higher integers, which can be used as indexes in a for loop.
  - Zip function - returns a series of parallel-item tuples, which can be used to traverse multiple sequences in a for loop.
  - Enumerate function - generates both values and indexes of items in an iterable so we don't need to count manually.
  - Map function - calls a function on each item in an iterable
  - Filter function - returns items in an iterable for which a passed in function returns true

List Comprehension

- List comprehensions are written in square brackets(‘[]’) because they are ultimately a way to construct a new list.
  - They begin with an arbitrary expression.
  - Followed by a for loop header.

```
>>> L
[21, 22, 23, 24, 25]
```

Using Loops

```
>>> res = []
>>> for x in L:
...     res.append(x + 10)
...
>>> res
[31, 32, 33, 34, 35]
```

produces the same results

List Comprehension

```
L = [x + 10 for x in L]
```

- x iterates through L, and on each iteration the expression (x+10) is applied. The result is appended to a list.

List Comprehension

- List comprehension is never required. for loops can accomplish the same tasks.

However,

- List comprehensions are more Pythonic
- List comprehensions are optimized and can achieve up to double the speeds of equivalent for loops

List Comprehension

List comprehension can be extended and made more complex.

- We can create a filter clause using an if clause.
  - Per for clause we can have one if statement.

```
>>> lines = [line.rstrip() for line in open('script2.py') if line[0] == 'p']
>>> res = []
>>> for line in open('script2.py'):
...     if line[0] == 'p':
...         res.append(line.rstrip())
```

- We can nest more for clauses.

```
>>> [x + y for x in 'abc' for y in 'lmn']
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']

>>> res = []
>>> for x in 'abc':
...     for y in 'lmn':
...         res.append(x + y)
```

Setting up Atom IDE

Download @ https://atom.io/

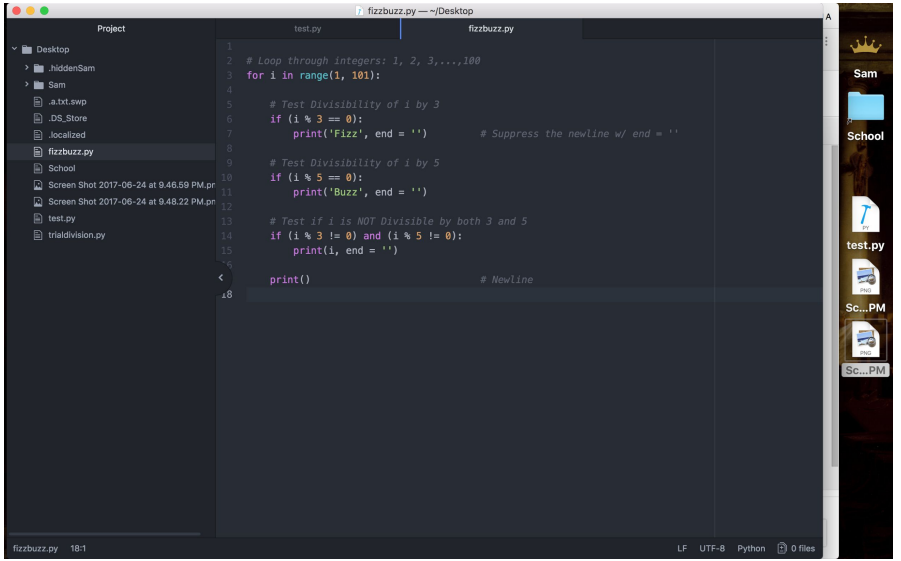
Installation Guide @

http://flight-manual.atom.io/getting-started/sections/installing-atom/#platform-mac
http://flight-manual.atom.io/getting-started/sections/installing-atom/#platform-windows
http://flight-manual.atom.io/getting-started/sections/installing-atom/#platform-linux

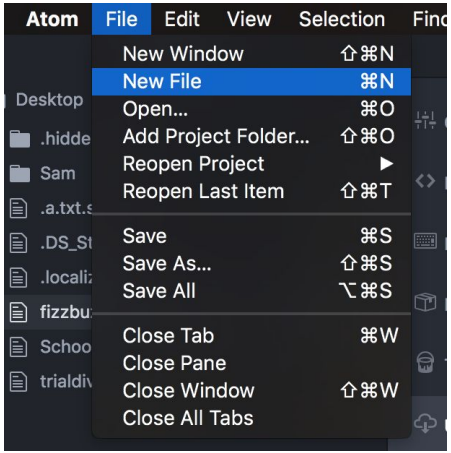
List of Other Python IDEs:

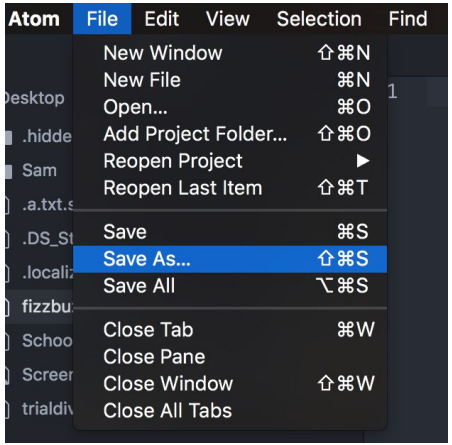
https://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments#Python

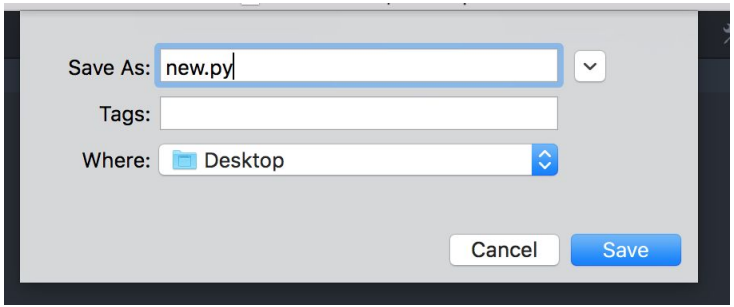
Important Features:
>Syntax highlighting
>Indentation
>Autocomplete
>Brace matching



Creating a New .py File



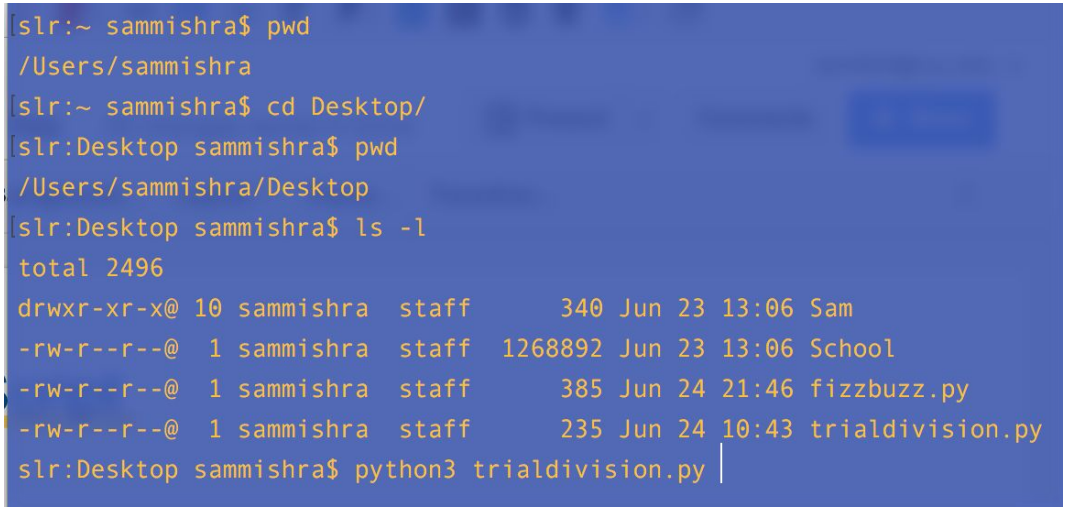




Running a .py Script

1) Navigate to the directory containing the .py script you wish to run
2) Run the script with this command:

python[3] yourscript.py



Programming

The Zen of Python

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

Programming

- Wrap lines so that they don’t exceed 79 characters.
  - This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.
- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own.
- Use docstrings.
- Use spaces around operators and after commas, but not directly inside bracketing constructs: `a = f(1, 2) + g(3, 4)`.
- Name your classes and functions consistently;
  - the convention is to use `CamelCase` for classes and
  - `lower_case_with_underscores` for functions and methods.

Programming

Algorithm

An algorithm is a step by step process that describes precisely how to solve a problem and/or complete a task, which will always give the correct result.

Pseudo-Code

Matches a programming language fairly closely, but leaves out details that could easily be added later by a programmer.
Pseudocode doesn't have strict rules about the sorts of commands you can use, but it's halfway between an informal instruction and a specific computer program.

Programming - Fizz Buzz

Write a program that prints the numbers from 1 to 100.

But for multiples of three print “Fizz” instead of the number

And for the multiples of five print “Buzz”.

For numbers which are multiples of both three and five print “FizzBuzz”."

Programming - Fizz Buzz Solution

```python
# Loop through integers: 1, 2, 3,...,100
for i in range(1, 101):

    # Test Divisibility of i by 3
    if (i % 3 == 0):
        print('Fizz', end = '')          # Suppress the newline w/ end = ''

    # Test Divisibility of i by 5
    if (i % 5 == 0):
        print('Buzz', end = '')

    # Test if i is NOT Divisible by both 3 and 5
    if (i % 3 != 0) and (i % 5 != 0):
        print(i, end = '')

    print()                              # Newline
```

```
slr:Desktop sammishra$ python3 fizzbuzz.py
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
Fizz
22
23
Fizz
Buzz
```

This is not the only correct way to code a solution to this problem

Programming - Sieve of Eratosthenes

The **sieve of Eratosthenes** is a simple, ancient algorithm for finding all prime numbers up to any given limit.

1. Create a list of consecutive integers from 2 through n: (2, 3, 4, ..., n).
2. Initially, let p equal 2, the smallest prime number.
3. Enumerate the multiples of p by counting to n from $2p$ in increments of p, and mark them in the list (these will be $2p$, $3p$, $4p$, ...; the p itself should not be marked).
4. Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from step 3.
5. When the algorithm terminates, the numbers remaining not marked in the list are all the primes below n.

Task: Write a script that implements the Sieve of Eratosthenes.

Programming - Sieve of Eratosthenes

```python
# N - We will find all primes less than N
N = 100

# Create three empty lists
# - One to collect primes
# - One to collect the numbers we haven't checked yet
# - One to collect the composites
primes = []
numbers = []
composites = []

# Initialize the numbers list
for i in range(2,N+1):
    numbers.append(i)

# Loop while there are unchecked numbers
while len(numbers) > 0:

    # The first number in the numbers list is prime
    primes.append(numbers[0])

    # Any multiple of that first number is a composite
    for i in range(numbers[0],N+1,numbers[0]):
        composites.append(i)

    # Use sets to remove the composites
    numbers = list(set(numbers)-set(composites))

    # Sets are unorderd, so numbers must be put back in order
    numbers.sort()

# Print the prime list
print(primes)
```