STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

SSW-500: Introduction to Software Development

Python Fundamentals

Dr. Nafiseh Ghorbani Renani
Software Engineering
School of Systems and Enterprises

This Lecture is based on material from Prof. Jim Rowland

What this course is about?

- This hands-on course introduces the student to Object Oriented analysis, design, and programming with Python and covers several related software development tools and techniques that are critical for success as a member of software engineering team.
- The course begins with an introduction to Object Oriented Design and programming with Python. We also explore collaborative source code and project management with GitHub, along with software analysis, debugging, testing, and refactoring techniques. Finally, relational and NoSQL database technologies are also introduced.

Quizzes, Homework and Project

- Short online quiz covering the reading and lecture material
  - To help you retain the material
- Homework assignment each week (subject to change)
  - You may ask me questions about the homework before turning it in, but all work must be your own
  - Upload your homework answers to Canvas
- The course project: hands-on experience on solving real world problems; the course projects will be developed over the second half of the course. Group work is not required (2 team members maximum).

Be sure to answer all questions in your own words---don't copy from the lecture notes or other sources.

Copy and paste from lecture or web ***is not acceptable***

Social Media

- How many hours do you spend on Social Media each day?
- Reallocate time for enhancing your career
  - New technologies
  - New approaches, tools, and techniques
  - Pulse of the industry
- Bi-weekly assignments to share an article of your choice

Cheating

- ***Cheating will NOT be tolerated***
- ALL work is expected to be in your own words
- all quiz answers
- all homework answers
- all exam answers
- all programming
- Copying from any source is considered cheating
- providing a citation does not excuse this
- Consequences of cheating may include:
- receiving a grade of 0 for an assignment
- receiving a grade of "F" for the course
- expulsion from the university

Acknowledgements

This lecture includes material from

"How to Think Like a Computer Scientist: Learning with Python 3 (RLE)", Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers,

http://openbookproject.net/thinkcs/python/english3e/

And

"Introducing Python", Bill Lubanovic, ISBN-13: 978-1449359362

http://proquest.safaribooksonline.com/book/programming/python/9781449361167

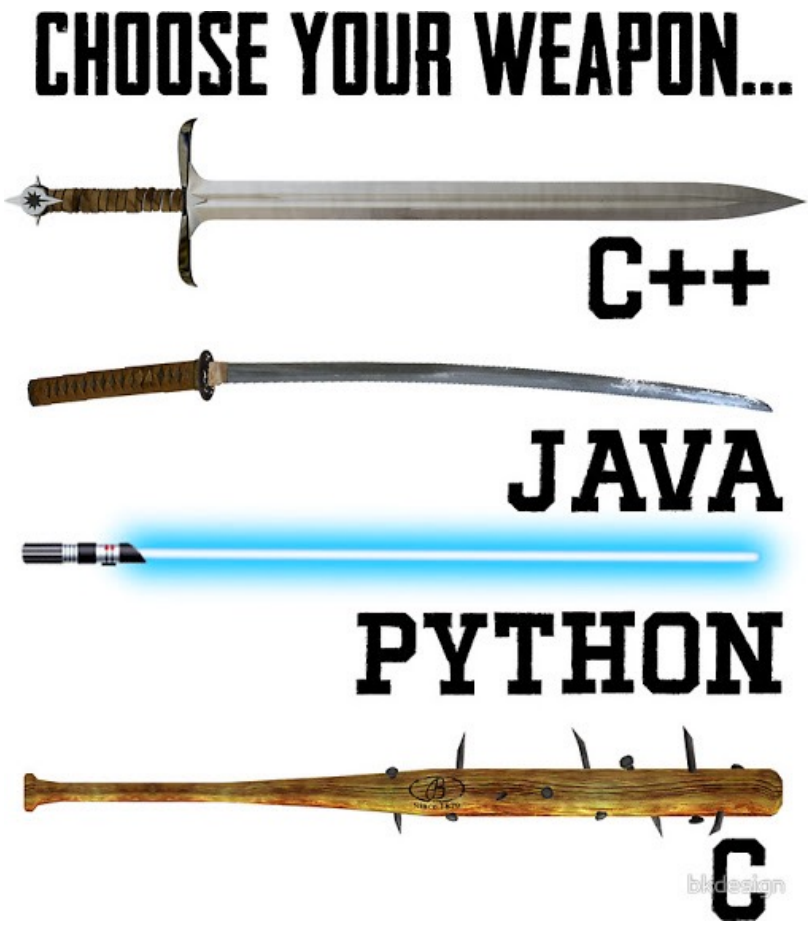
Today's topics

Python Overview

- Variables, statements, expression
- Operators/Operands
- Functions
- *Booleans, conditionals*
- *If/elif/else*
- *While/for*
- *List, dict* containers

Integrated Development Environments (IDE)

Just for fun…



Talking to Python

Say hello!

```
print('hello world')
```

You type the stuff in the pink/grey box

```
hello world
```

Output from Python

Python is an interpreted language

- Commands are parsed and executed interactively
- No need to compile

If you're unsure of what Python would do, try it!!!

Designing Python experiments is a critical skill

Comments: Note to self!

Comments allow us to leave reminders for us and others who read our code

```
# this is a comment
# Python ignores comments
print('Hello world!')
```

```
Hello world!
```

```
""" This is a docstring """
''' this is also a docstring '''
```

Comments are ***critical*** for readable, maintainable code

Code lives far longer than we expect, and comments help us to understand what we were thinking after the fact

Python primitives (Py ingredients)

```
2
3.14157
'Hello world!'
"Hello world!"
"I'm a string with an apostrophe"
'A string with "double" quotes'
""" Docstrings are strings, but also statements """
```

An integer number

A floating point number

Two equivalent ways to specify a string

Double quotes make it easy to include apostrophes

Dynamic typing

Values have ***types*** that are determined dynamically

You can ask Python to identify the type of a value

```
type(2)
```

```
int
```

```
type(3.14157)
```

```
float
```

```
type('hello world')
```

```
str
```

```
type('2')
```

```
str
```

What? Why not an int???

Optional Type Hints

Python 3.6+ allows optional type hints to specify the expected type of variables and functions

```
an_integer: int = 2
a_float: float = 3.14
a_str: str = "hello world"
```

```
def plus1(n: int) -> int:
    return n + 1
```

Plus1 expects to be passed one int and returns an int

```
an_integer = "goodbye"
```

What??? We defined 'an_integer' to be an int

Python **ignores** the type hints at run time

IDEs use the the type hints to warn about potential problems

Optional Type Hints

Type hints specify the data type of variables, parameters and the value returned by functions

If you need …	Example	Use type:
string	`"hello world"`	`str`
integer	`42`	`int`
float	`3.14`	`float`
integer or float	`42 or 3.14`	`float`
Boolean	`True, False`	`bool`
Function returns None or no value		`None`

We'll look at many other types, including user defined types

Why specify type hints?

Type hints help to document the code and automatically detect problems

Users > jrr > Downloads > why_type_annotations.py > ...

```
1   def factorial(n: int) -> int:
2       """ return n! """
3       print(f"calculating factorial({n})")
4       if n == 1:
5           return 1
6       else:
7           return n * factorial(n - 1)
8
9   print(f"3! = {factorial(3)}")  # valid call to compute 3!
10  print(f"'3'! = {factorial('3')}")  # invalid call: should pass int, but passed str
11  print(f"3.1! = {factorial(3.1)}")  # factorial(float) may cause stack overflow if n never == 1
12
```

Factorial() expects one int and returns an int

PROBLEMS 2 OUTPUT DEBUG CONSOLE ... Filter. E.g.: text, **/*.ts, !**/node_modules/**

why_type_annotations.py ~/Downloads 2

- Argument 1 to "factorial" has incompatible type "str"; expected "int" mypy(error) [10, 28]
- Argument 1 to "factorial" has incompatible type "float"; expected "int" mypy(error) [11, 28]

Identified two problems without running the code!

Python keywords (reserved words)

Python reserves some words that can't be used as names

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

Python statements

Statements allow us to tell Python what we want to do

Examples of statements

```
cnt: int = 42

while cnt < 50:
      cnt += 1

if cnt == 51:
    print(51)

for item in ['donut', 'bagel']:
    print("I'm hungry for a", item)
```

```
I'm hungry for a donut
I'm hungry for a bagel
```

The Python interpreter (REPL) evaluates statements

- Statements don't produce results
- Python interpreter doesn't print anything after statements

Python expressions

Expressions are combinations of values, variables, and function calls

```
cnt: int = 42
```

This is a statement so no output

```
cnt
```

Evaluate the expression cnt

```
42
```

```
len("Hello world")
```

Call the `len` function

```
11
```

```
3 * 4 + 5
```

Evaluate the expression

```
17
```

Assignment statements

```
n: int = 42
n = n + 1
```

Python provides syntactic short cuts to improve readability

Statement	Equivalent statement
n = n + 1	n += 1
n = n - 1	n -= 1
n = n / 2	n /= 2
n = n * 2	n *= 2
n = n % 2	n %= 2
n = n // 10	n //= 10

Wait! What's that? How can we discover the semantics of '%'?

Python does ***not*** support `n++, ++n, n--, --n`

Getting input from the user

Python provides a function to get input from a user

```
response: str = input("Please enter a number:")
```

```
Please enter a number:42
```

```
response
```

```
'42'
```

Input always returns a string

```
response + 1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-d35ac03d859c> in <module>
----> 1 response + 1

TypeError: can only concatenate str (not "int") to str
```

Python is unhappy ;-(

```
int(response) + 1
```

Cast strings to integers or floats to do math

```
43
```

Printing output to the user

Python's print function writes output to the user

```
name: str = "Jim"
number: int = 42
print(f"My name is {name} and my favorite number is {number}")
```

f"… {expr} …" replaces {expr} with the value of {expr} in the formatted string

```
My name is Jim and my favorite number is 42
```

```
print("My name is", name, "and my favorite number is", number)
```

```
My name is Jim and my favorite number is 42
```

Print accepts any number of arguments of many types

Don't forget the 'f'
f"… {expr} …"

Functions

Functions are a sequence of statements

```
def name(parameters) -> return_type:
    """ parameters has parameter_1: type_1, parameter_2: type_2, ... """
    statements
```

Where:

`name`: the function name which must be a legal identifier

`parameters`: 0 or more identifiers separated by commas. These parameters are available as variables inside the function

`return_type:` the type returned by the function

`statements`: one or more statements

Functions may include an optional `return` statement

Functions ***always*** return a value (**None** by default)

Functions: *hello*

"**Parameters**" in the function definition can be used in the function definition

```
def hello(name: str) -> str:
    return f"Hello {name}!"
```

```
hello("Nanda")
```

```
'Hello Nanda!'
```

`Name` has value "`Nanda`" inside function `hello()`

Values passed into functions are called "**arguments**"

Functions: n^2

"**Parameters**" in the function definition

```
def square(n: int) -> int:
    return n * n
```

The parameters are available as variables inside the function

```
square(2)
```

n has value 2

```
4
```

Functions: n^2

What if we want calculate square(int) and square(float)?

```
def square(n: float) -> float:
    return n * n

square(3.14)
```

```
9.8596
```

float type covers both float and int

```
square(2)
```

```
4
```

One function (and type hint) handles both float and int

Function definitions

Python uses whitespace to define blocks

- Use whitespace consistently
- 4 spaces is standard to optimize readability
- Most IDEs automatically convert tab to 4 spaces

```
def name(parameters) -> return_type:
    statement_1
    statement_2
    statement_n
```

4 spaces

Python whitespace

You must consistently indent or Python won't understand your function definition

Some languages use { … } e.g. C, C++, Java
Python uses indenting to define related blocks of code

Use 4 spaces to ident

```
def factorial(n: int) -> int:
    """ return n! == n*n-1*n-2*... """
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

No semi-colons or curly braces in Python

Indent carefully to avoid frustration!

Documenting functions

Every file and function should begin with a **docstring**

```
def factorial(n: int) -> int:
    """ return n! == n*n-1*n-2*... """
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

docstring

```
help(factorial)
```

A function's docstring is available from the help() function

```
Help on function factorial in module __main__:

factorial(n: int) -> int
    return n! == n*n-1*n-2*...
```

Fruitful functions

Every **function returns a value:**

- `return` statement returns an explicit value
- Falling out the bottom of the function returns `None`

Not all functions calculate a value to return to the caller

```
print('Hello world')
```

- Print returns `None` but we don't use the return value or care

Functions that return a useful value are called ***fruitful*** functions

Some functions exist for their side effects, e.g. printing

Your turn

What does this function return?

```
def foo(n: int) -> int:
    print('Called foo')
    n + 1

result = foo(3)
result = ???
```

What is result?
3? No
4? No
None? Yes
No explicit return returns None

Recursive Functions

A python functions can call other functions

Including itself (recursion)

4! = 4 * 3 * 2 * 1

3! = 3 * 2 * 1

2! = 2 * 1

1! = 1

```
def factorial(n: int) -> int:
    """ return n! == n*n-1*n-2*... """
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Boolean values

Python includes a Boolean type (`bool`)

A **Boolean** value is either ***True*** or ***False***

```
5 < 8
```

```
True
```

```
5 > 8
```

```
False
```

```
5 == (3 + 2)
```

```
True
```

Functions can return True/False

It's sometimes useful to define functions that return True/False for use as Boolean operands

```
def is_even(n: int) -> bool:
    return n % 2 == 0

def is_odd(n: int) -> bool:
    return n % 2 == 1
```

```
is_even(42)
```

```
True
```

```
is_odd(42)
```

```
False
```

Comparison operators

Conditionals help to make decisions

Comparison Operator	Semantics	Example
==	Equal	`3 == 4`
!=	Not equal	`3 != 4`
<	Less than	`'a' < 'b'`
>	Greater than	`'a' > 'b'`
<=	Less than or equal to	`3 <= 4`
>=	Greater than or equal to	`3 >= 4`

Logical operators

Combine logical expressions with logical operators:

`and, or, not`

```
5 < 8 and 10 > 12 # both must be True
```

```
False
```

```
5 > 8 or 7 < 9 # only one must be True
```

```
True
```

```
not 3 < 2
```

```
True
```

Truth tables for logical operators

x	y	x and y[1]	x or y[2]
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

[1] Both must be True or `x and y` is False

[2] Only one of x, y must be True for `x or y` to be True

Short circuiting

Python evaluates logical expressions from left to right

If the ***left*** operand of an `or` operator is True then the right operand is ***not*** evaluated and the expression is True

- If the ***left*** operand of an `or` operator is False then both operands are evaluated and the result depends on the right operand

If the ***left*** operand of an `and` operator is False then the right operand is not evaluated

- If the ***left*** operand of an `and` operator is True then both operands are evaluated and the result depends on the right operand

Short circuiting: `or` (disjunctions)

```
def is_true() -> bool:
    print('called is_true()')
    return True

def is_false() -> bool:
    print('called is_false()')
    return False
```

```
is_true() or is_false()
```

```
called is_true()

True
```

is_false() is **not** called

```
is_false() or is_true()
```

```
called is_false()
called is_true()

True
```

Both is_false() and Is_true() are called

Short circuiting: `and` (conjunctions)

```
def is_true() -> bool:
    print('called is_true()')
    return True

def is_false() -> bool:
    print('called is_false()')
    return False
```

```
is_true() and is_false()
```

```
called is_true()
called is_false()

False
```

Must call both functions to determine if the expression is True/False

```
is_false() and is_true()
```

```
called is_false()

False
```

Only first function must be called to know that the expression is False

if .. then … elif ... else statement

```
if condition_1:
    statement_1_1
    statement_1_2
elif condition_2:
    statement_2_1
    statement_2_2
else:
    statement_3_1
    statement_3_2
```

If condition_1 is True, then execute the statements in this block

If condition_1 is False and condition_2 is True, then execute the statements in this block

If all previous conditions are false, then execute the statements in this block

0 or more `elif` statements

0 or 1 `else` statements

Nested if statements

```
1  home_state: str = 'NJ'
2  age: int = 16
3
4  if home_state == 'NJ':
5      if age >= 17:
6          print("Eligible for car license")
7      else:
8          print("Maybe next year")
9  elif home_state == "IA":
10     if age >= 16:
11         print("Eligible for car license")
12     elif age >= 14:
13         print("Eligible for tractor license")
14     else:
15         print("Maybe next year")
```

```
Maybe next year
```

Which `else` goes with which `if`?

- Indentation defines the statement blocks
- Careful indentation is ***critical***!

Python does ***not*** have a case/switch statement

Your turn

What is the output if x == 3?

```
if x <= 4:
    if x <= 2:
        if x == 1:
            print('1')
        else:
            print('2')
    elif x == 3:
        print('3')
    else:
        print('4')
elif x == 5:
    print('5')
else:
    print('> 5')
```

McCabe's Cyclomatic complexity measures the number of potential paths through if statements.

Empirical studies show that functions with more than about 10 different paths are too hard to maintain.

Keep the number of alterative paths small to improve readability and maintainability.

Conditional one liners

```
n: int = 42
result1: str = ""

if n >= 0:
    result1 = 'Positive'
else:
    result1 = 'Negative'
```

We can collapse a conditional into a single line

True_result `if` ***Conditional_expr*** `else` ***False_result***

```
result2: str = 'Positive' if n >= 0 else 'Negative'
```

This is known as the ***ternary operator***

Code may be more concise, but possibly harder to read

`return` – Get me out of here

Functions normally return after the last statement, but you may want to return from the middle of the function

```
def print_square_root(n: int) -> None:
    if n < 0:
        print('Only positive numbers, please')
        return

    result: float = n ** 0.5
    print(result)
```

Careful with early return from functions

- May make functions harder to read and understand
- Early return may be best solution in some cases

`return` – Don't let this happen to you

```
def absolute_value(n: float) -> float:
    result: float
    if n < 0:
        result = -1 * n
        return result
    else:
        result = n
```

`float` handles both `int` and `float`

What's returned if n >= 0?

`result`?

No! No explicit `return` statement returns `None`

Insure that all exit paths have `return` statements

`return` – Don't forget me!!!

What's wrong with this code?

```
def this_cant_be_good():
    do_something_useful_1()
    return
    do_something_useful_2()
    do_something_useful_3()
```

These statements will never be executed

Be alert for unreachable statements

Static analysis tools can help to find these problems

while loops

```
while conditional:
    statement1..n
statement n+1
```

Execute $statement_{1..n}$ while conditional is True

```
n: int = 0
while n < 2:
    print(n)
    n += 1
print('done')
```

Verify that the Boolean expression eventually becomes False to avoid infinite loops

```
0
1
done
```

Write a function…

Write a function to calculate the sum of a range of integers

```
def sum_range_while(start: int, end: int) -> int:
    """ calculate the sum of the integers starting
        at start and up to, but not including, end
    """
    total: int = 0
    while start < end:
        total += start
        start += 1
    return total

assert sum_range_while(1,1) == 0
assert sum_range_while(1,5) == 10
assert sum_range_while(0,2) == 1
```

What if start >= end?

`assert expr` complains if expr is not True

`break` out

Sometimes you want to break out of the middle or end, rather than the top, of a loop

```
while conditional:
    statements
    if condition:
        break
    statements
statement
```

`break` leaves the loop and execution moves to the first line following the while block

do while?

Python does ***not*** include a do while statement…

… but it's easy to simulate

```
while True:
    statements
    if condition: # test to terminate loop
        break
statement
```

`for` loops

`for` loops iterate through sequences

```
for item in ['bagel', 'sandwich', 'coffee']:
    print(f"I'm hungry for a {item}")
```

```
I'm hungry for a bagel
I'm hungry for a sandwich
I'm hungry for a coffee
```

Python has many different sequences, e.g. lists, dicts, tuples, strings, ranges, generators, …

`for (i = 0; i < 3; i++){ }`

`for` loops iterate through sequences

What about a range of 0 to *n*?

```
for i in range(3):
    print(i)
```

```
0
1
2
```

`range(limit)` returns a sequence of integers from `0` ***up to***, but ***not*** including, `limit`

ADVANCED/OPTIONAL

Slides marked **ADVANCED/OPTIONAL**
contain topics that we'll cover in detail later in the course

```
if comfortable_with_advanced_topics():
    use_in_homework_assignments()
elif confused():
    don't_worry_for_now()
    use_simpler_approach()
```

Python lists

ADVANCED/OPTIONAL

Lists hold arbitrary, ordered values of different types

```
1 from typing import List, Any
2 lst: List[Any] = ['coffee', 42]
3 lst[0]
```

```
'coffee'
```

Create a list

Access individual elements with slices offset from 0

```
1 lst[0] = 'tea'
2 lst[0]
```

```
'tea'
```

Change the list with slices

```
1 lst.append(3.14)
```

Append items to the end of the list

```
1 for item in lst:
2     print(item)
```

Iterate through the elements in the list

```
tea
42
3.14
```

Type hints for lists ADVANCED/OPTIONAL

Lists hold arbitrary, ordered values of different types

If all elements in the list have the same type, then specify that type in the type hint

Import type hints from `typing`

Type of values in the list

```
from typing import List, Any

primes: List[int] = [1, 2, 3, 5, 7, 11]

radius: List[float] = [1.1, 2.2, 3.3]

names: List[str] = ['Nanda', 'Maha', 'Fei']

values: List[Any] = [1, 2.3, ['a', 4, ['b', 5]]]
```

Finding an item in a collection

ADVANCED/OPTIONAL

I need to find my friend Nanda

I can knock on every door until I find him …

… or I can look up his address and go directly to the right door

Dictionaries map keys to values

Nanda ⇨ Apt 5B

Python dict maps a key to a value

ADVANCED/OPTIONAL

```
translate:Dict[str, str] = {'one': 'uno', 'two': 'dos'}
translate['two']
```

```
'dos'
```

Key

Value

Initialize a dict

Use key to retrieve associated value

```
translate['three'] = 'tres'
```

Add new key/value pairs

```
print("English", "Spanish")
for key, value in translate.items():
    print(key, "\t", value)
```

```
English Spanish
one     uno
two     dos
three   tres
```

Iterate through key/value pairs

The order is undefined…

Type hints for dicts

ADVANCED/OPTIONAL

Lists hold arbitrary, ordered values of different types

If all elements in the list have the same type, then specify that type in the type hint

`var:Dict[`*key_type*`,` *value_type*`]`

Type of the key

Type of the value

```
from typing import Dict, Tuple

english2spanish:Dict[str, str] = {'one': 'uno', 'two': 'dos'}

int2str:Dict[int, str] = {1:'one', 2:'two', 3: 'three'}

str2int:Dict[str, int] = {'one': 1, 'two': 2, 'three': 3}

rooms: Dict[Tuple[str, str], str] = {('SSW', '810A'): "GN 204"}
```

Typical Python Program Files

```
1   """
2       This file demonstrates a typical Python file
3   """
4   from typing import List, Tuple, Dict
5
6   def say_hello(whom: str) -> str:
7       """ return a string of "Hello {whom}!" """
8       return f"Hello {whom}!"
9
10  def main() -> None:
11      """ Say hello to a few friends """
12      print(say_hello("Nanda"))
13      print(say_hello("Maha"))
14
15  if __name__ == "__main__":
16      main()
```

Docstring to describe the file

Imports go at the top of the file

Functions with docstrings and logic

main() function with test cases and user interaction

Call main()

__name__ identifies current module

"__foo__" specifies something magic in Python

ADVANCED/OPTIONAL

file1.py

```
print(f"file1.py: __main__='{__name__}'")
if __name__ == "__main__":
    print(f"file1.py: running as main")
else:
    print(f"file1.py: being imported")
```

$ python file1.py

```
file1.py: __main__='__main__'
file1.py: running as main
```

__name__ specifies the name of the current module

file2.py

```
print(f"file2.py: __name__='{__name__}'")

import file1

if __name__ == "__main__":
    print(f"file2.py: running as main")
else:
    print(f"file2.py: being imported")
```

$ python file2.py

```
file2.py: __name__='__main__'
file1.py: __main__='file1'
file1.py: being imported
file2.py: running as main
```

Is that all?

Python has many more features that we'll explore throughout the semester but this gives you enough to get started on your journey to becoming a Pythonista…

Coding style guidelines

Code lives forever so make it readable and maintainable

Good comments are critical

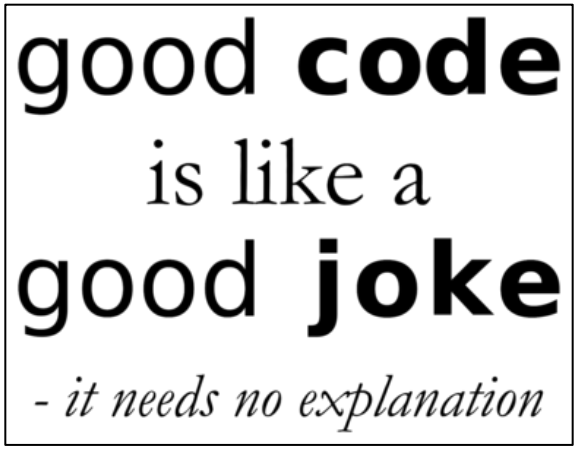
- Don't explain the obvious but do explain subtleties
- Include a comment at the top to explain the problem
- Explain major sections

Use meaningful variable names

Use consistent white space/blanks

```
var: type = value
```

- Blank lines between sections



We'll explore more formal guidelines later

Integrated Development Environments

You're ready to write Python code!

Alternatives for interacting with Python

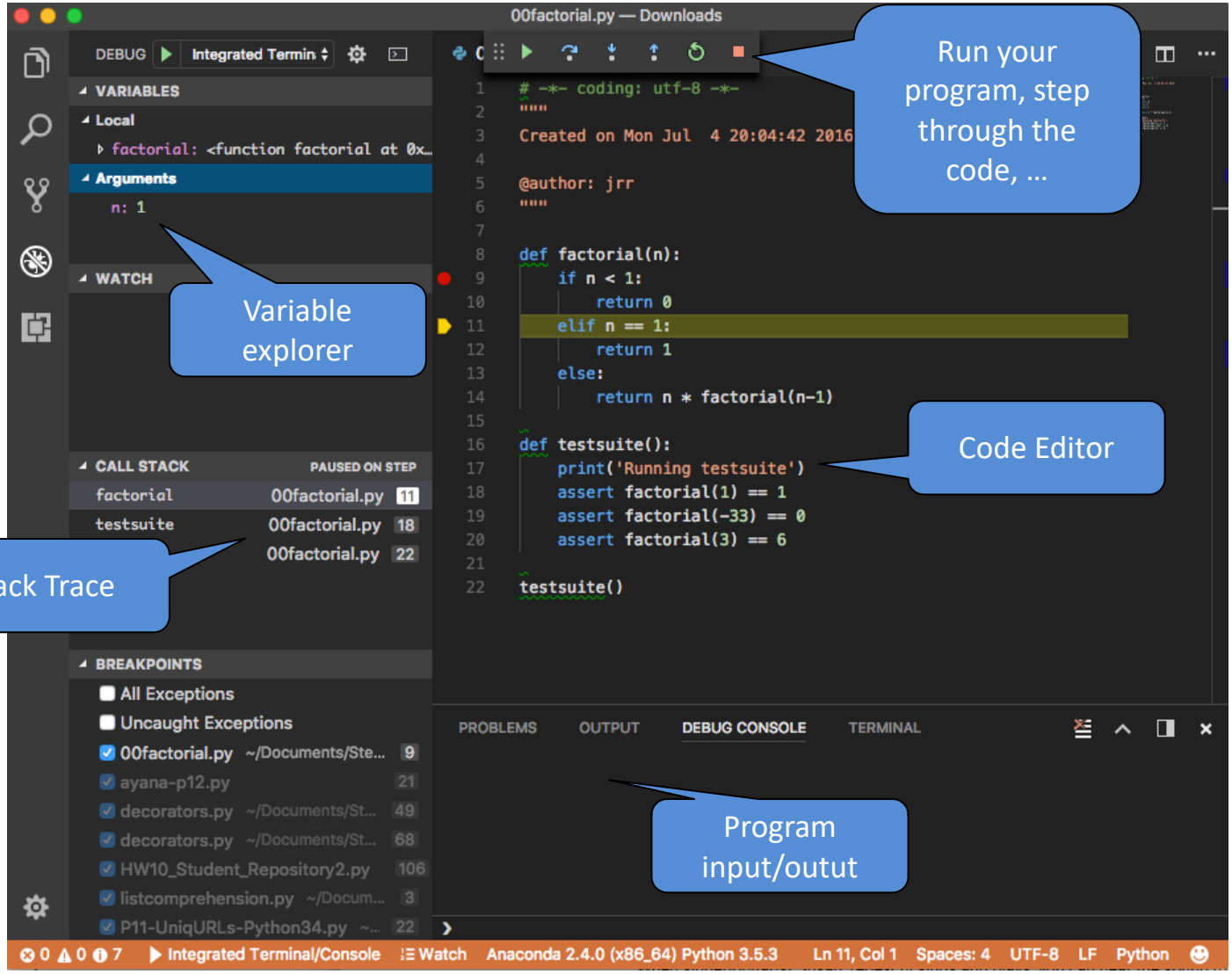
- Type into the Python interpreter
- Create a .py file in a text editor
- Use an Integrated Development Environment (IDE)

Your choice, but an IDE will be easiest and most efficient after a short learning curve

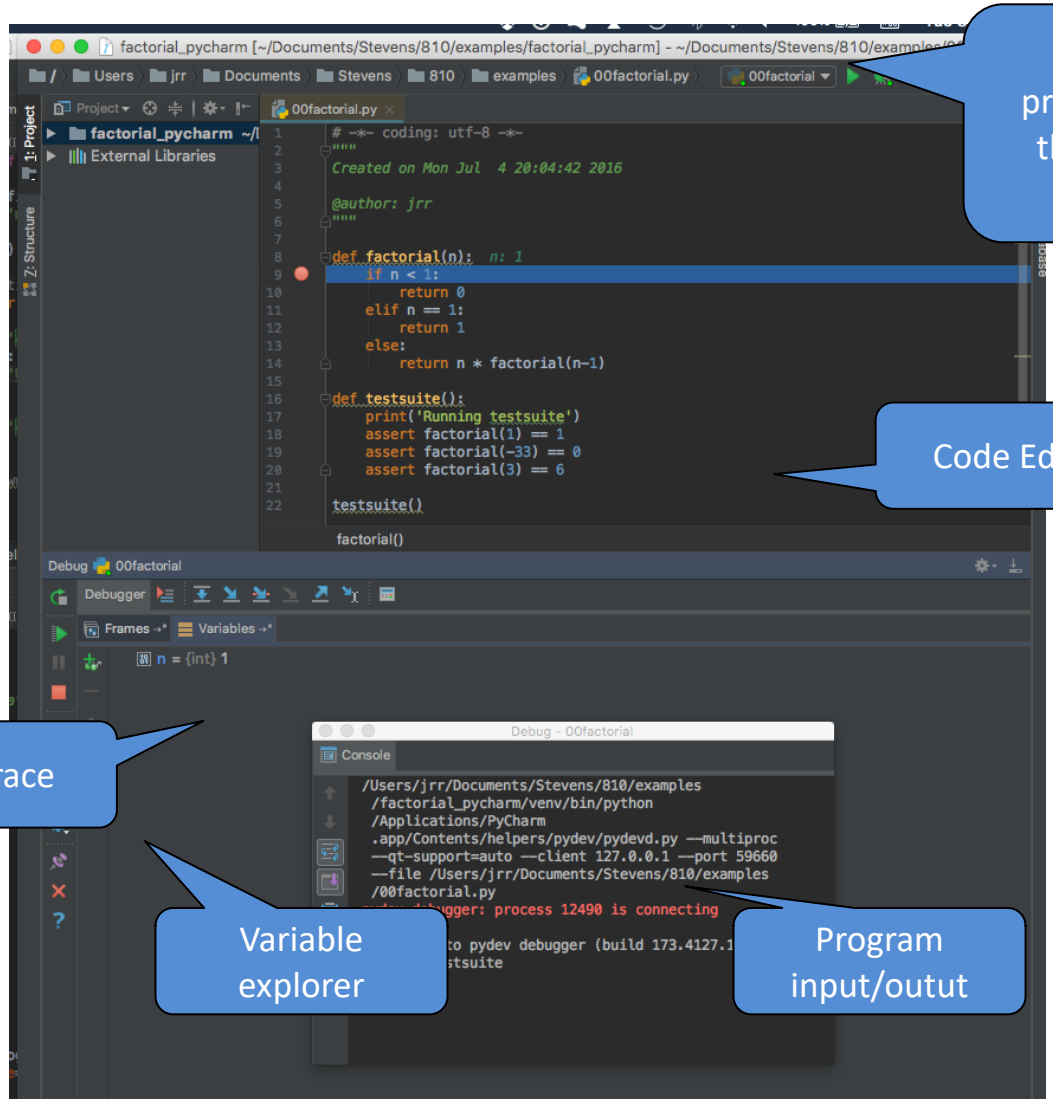
Many IDEs available

- VS Code, PyCharm, Jupyter, PyScripter, IDLE, …

Visual Studio Code



PyCharm



Run your program, step through the code, …

Code Editor

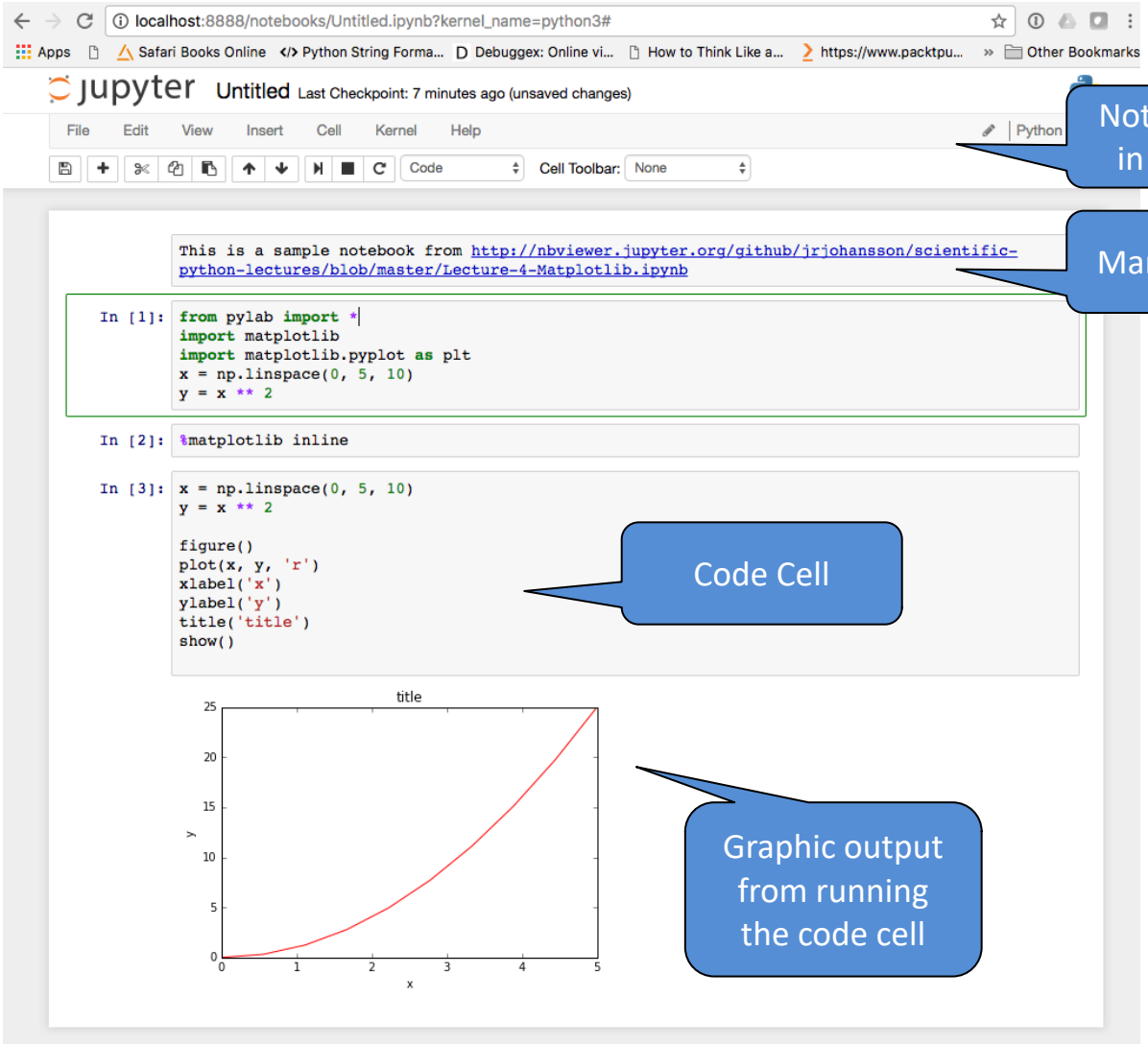
PyCharm is VERY powerful and includes many powerful features but may be harder to get started

Stack Trace

Variable explorer

Program input/outut

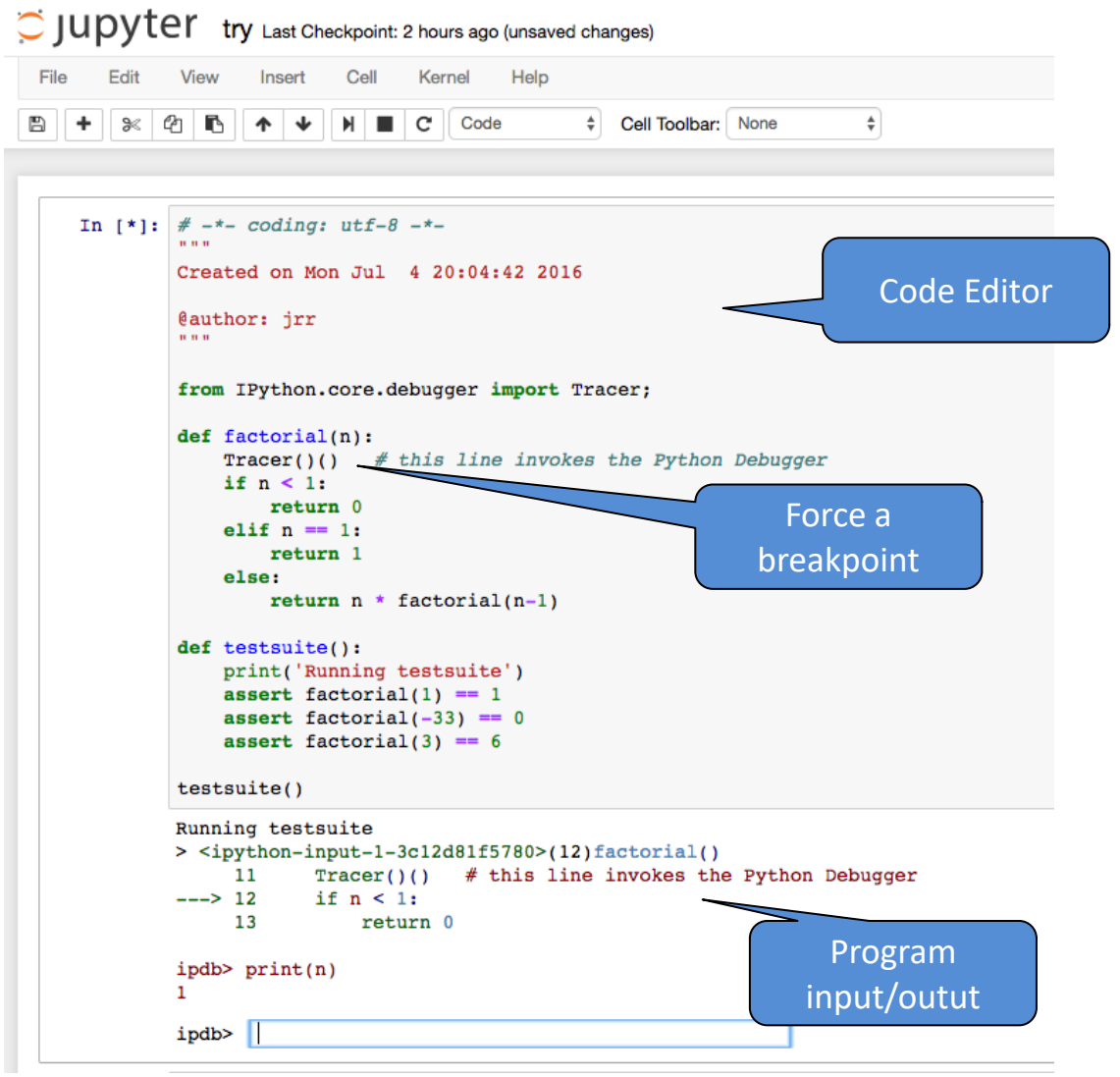
Jupyter Notebook



Jupyter is very popular for Data Science because you can combine code, Markup, output, etc.

Describe the logic and results of the analysis in a single, executable notebook

Debugging Jupyter Notebooks



Debugging may be less convenient, with command line ipdb/pdb debugger

IDE Summary

IDE Product	Advantages	Disadvantages
VS Code	Supports many languages, works well with Python	
PyCharm	Very powerful	Steeper learning curve than other IDEs
Jupyter Notebooks	Very popular choice for Data Science, exploration, and sharing results	Debugger is not tightly integrated and requires command line interface

It's ***your*** choice!
Pick one of these, or any other you prefer,
but **you must** use an IDE for this course

Questions?