An ALGORITHM (a structured solution to a problem) is a sequence of well-defined operations.

How do you create a good one?
Sam Schwartz

- PhD Student
Sam Schwartz

- PhD Student
Sam Schwartz

- PhD Student
- Taught HS English in Chile, worked as a software monkey for Amazon.
REVIEW ALL MATERIAL AT THE COURSE WEBSITE!
REVIEW ALL MATERIAL AT THE COURSE WEBSITE!

Required text is on reserve in library.
REVIEW ALL MATERIAL AT THE COURSE WEBSITE!

Required text is on reserve in library.

Schedule page – Weekly topic and readings, Class notes, Lab exercises, Projects, Project Solutions, Practice Exams, Exams, Solutions
Required text is on reserve in library.

**Schedule page** – Weekly topic and readings, Class notes, Lab exercises, Projects, Project Solutions, Practice Exams, Exams, Solutions

**Syllabus** – Class policies, how final grade is calculated, much more
REVIEW ALL MATERIAL AT THE COURSE WEBSITE!

Required text is on reserve in library.

**Schedule page** – Weekly topic and readings, Class notes, Lab exercises, Projects, Project Solutions, Practice Exams, Exams, Solutions

**Syllabus** – Class policies, how final grade is calculated, much more

**Other Information** – Lab help hours schedule, contact information, helpful and interesting links
REVIEW ALL MATERIAL AT THE COURSE WEBSITE!

Required text is on reserve in library.

**Schedule page** – Weekly topic and readings, Class notes, Lab exercises, Projects, Project Solutions, Practice Exams, Exams, Solutions

**Syllabus** – Class policies, how final grade is calculated, much more

**Other Information** – Lab help hours schedule, contact information, helpful and interesting links

classes.cs.uoregon.edu/20W/cis210/
classencore.uoregon.edu
210, 211, 212 Computer Science I, II, III
prereqs:  - Math 112 (231, 251 readiness)
          - prior programming experience
210, 211, 212 Computer Science I, II, III
prereqs:  - Math 112 (231, 251 readiness)
          - prior programming experience

familiar with programming fundamentals – variables, expressions, basic data types (e.g., numeric, boolean, string, collections), conditionals, loops, user-defined functions, parameter passing, ...
210, 211, 212 Computer Science I, II, III
prereqs:  - Math 112 (231, 251 readiness)
          - prior programming experience

familiar with programming fundamentals – variables, expressions, basic data types (e.g., numeric, boolean, string, collections), conditionals, loops, user-defined functions, parameter passing, ...

familiar with process of programming – time, bugs, technical glitches, multiple drafts ...
Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.

Douglas Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid*
Prerequisite Knowledge Check
What will be the result when the following code is executed?

def testfunc(timeLeft, time):
    '''
    What does this function do?
    '''
    while timeLeft > 0:
        print(timeLeft)
        timeLeft = timeLeft - 1
    if time == 'am':
        print('Good morning, CIS 210!')
    return timeLeft

>>> testfunc(10, 'am')
def testfunc(timeLeft, time):  # [defining a function/parameters]
    
    What does this function do?  # [documenting code]
    
    while timeLeft > 0:  # [loops, expressions, Boolean type, number]
        print(timeLeft)  # [input/output, variables]
        timeLeft = timeLeft - 1  # [assignment, built in operators]

    if time == 'am':  # [conditionals]
        print('Good morning, CIS 210!!')  # [string data type]
        # [function execution may have side effects]
        return timeLeft  # [functions return values]

What will be the result when the following code is executed?

>>> testfunc(10, 'am')  # [calling/executing a function, argument passing]
An ALGORITHM (a structured solution to a problem) is a sequence of well-defined operations. How do you create a good one?
Recall: Computational Problem Solving
Recall: Computational Problem Solving

Phase 1

TASK/PROBLEM ↔ Computational Thinking ↔ EXAMPLES/SOLUTION/ALGORITHM/COMPUTATIONAL PROCESS
Recall: Computational Problem Solving

Phase 1

**TASK/PROBLEM** ↔ **Computational Thinking** ↔ **EXAMPLES/SOLUTION/ALGORITHM/COMPUTATIONAL PROCESS**

Phase 2

**ALGORITHM** ↔ **Design/Coding** ↔ **COMPUTER PROGRAM**
Recall: Computational Problem Solving

Phase 1

TASK/PROBLEM $\leftrightarrow$ Computational Thinking $\leftrightarrow$
EXAMPLES/SOLUTION/ALGORITHM/COMPUTATIONAL PROCESS

Phase 2

ALGORITHM $\leftrightarrow$ Design/Coding $\leftrightarrow$
COMPUTER PROGRAM

Phase 3

COMPUTER PROGRAM $\leftrightarrow$ Testing/Debugging $\leftrightarrow$
HIGH QUALITY COMPUTER PROGRAM
A computer program implements an algorithm (computational process) on a computer.

A computer program is (therefore) a set of instructions written in a language the computer can understand.
A computer program is (therefore) a set of instructions written in a language the computer understands. What kind of language is that???

0’s and 1’s?
Natural language?
Python (or any computer programming language) provides a notation (syntax, semantics) for writing out a computational process as a series of steps.
Python (or any computer programming language) provides a notation (syntax, semantics) for writing out a computational process as a series of steps.

- Formal
- Precise
- Unambiguous
Well-written code is measured by...

- Validity
Well-written code is measured by...

- Validity
- Readability
Well-written code is measured by...

- Validity
- Readability
- Fluidity
Well-written code is measured by...

- Validity
- Readability
- Fluidity

“Code is more often read than written.”
—Guido Van Rossum
A computer program is (therefore) a set of instructions written in a language the computer understands.

Compromise: Programming Languages

What kind of language is that???
0’s and 1’s?
Natural language?
The programming language we use (Python) provides a (relatively) high level of abstraction with the computer.
Computer levels of abstraction:
Computer levels of abstraction:

- project1.py
- project2.py
Computer levels of abstraction:

- Office suites
- Web browser
- Python interpreter
- Project 1 (project1.py)
- Project 2 (project2.py)
- Other applications
Computer levels of abstraction:

- office suites
- web browser
- Python interpreter
- project1.py
- project2.py
- other applications
- operating system (OS X, Linux, Windows, iOS, Android etc.)
Computer levels of abstraction:

- office suites
- web browser
- Python interpreter
- operating system (OS X, Linux, Windows, iOS, Android etc.)
- project1.py
- project2.py
- other applications
- hard drive
- monitor
- CPU
Python is a language AND Python is a program (translator/interpreter)
Why Python?
Why Python?

- Python is modern, high level language; widely used in many fields
- Accessible to entry level programmers and also for experts – like chess or tennis
- Interactive (interpreted), syntax-lite language – concentrate on problem-solving rather than the language itself; can easily test snippets of code
- Lots of built in functionality and support libraries ("batteries included")
- General purpose, multiple paradigm language and syntax support straightforward transition to C, C++, Java
- Popular, well-supported, good documentation and development environments.
Python/IDLE

- Downloads along with Python
- Simple **integrated development environment** (IDE)
- **Shell** for exploring Python and testing bits of code
- **Editor** supports Python program development
Python Quick Overview

- Python language
- Python interpreter (program)

→ What sorts of Python language input does the Python interpreter recognize/"understand"?
What sorts of **Python language** input does the **Python interpreter** recognize/“understand”? 

Python (and any programming language) 
-- keywords 
-- primitive elements 
-- identifiers
What are the...

- Keywords
- Primitive Elements
- Identifiers

What does this function do?

```python
def testfunc(timeLeft, time):
    # [input data is provided when function is called (executed)]
    msg = 'Good morning, CIS 210!'  # [sequential flow of control, variable]
    while timeLeft > 0:
        # [flow of control, loop, conditional]
        print(timeLeft)  # [display result]
        timeLeft = timeLeft - 1  # [variable assignment]

        if time == 'am':  # [flow of control, conditional]
            print(msg)
            print(msg)

    return timeLeft  # [return result of executing testfunc]
```

What will be the result when the following code is executed?

```python
>>> testfunc(10, 'am')
??
```
Python keywords, primitive elements, identifiers

Keywords such as def, :, =, while, if, return -

- structure code
- store/retrieve values
- indicate order statements are executed
- special operations

Keywords are fixed, i.e., part of the language. We can use but not create them.
Welcome to Python 3.6's help utility!
...

help> keywords
Here is a list of the Python keywords. Enter any keyword to get more help.

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

help>
→ Python ✓ keywords, primitive elements, identifiers

**Keywords** define the syntax and structure of the language. They are fixed, i.e., we use keywords but cannot create new keywords.

**Primitive elements** are basic **objects** of the language, for example, integers, strings, and functions.

They can be combined to create new elements.
Python keywords, primitive elements, identifiers

Keywords define the syntax and structure of the language. They are fixed, i.e., we use keywords but cannot create new keywords.

Primitive elements are basic objects of the language (they are “built in”), for example, string, integers, and functions.

They can be combined to create new elements.

Identifiers are the names given to Python objects like integers, strings, functions, etc. (e.g., testfunc, time, timeLeft, print).

Some identifiers come with the language (they are “built in”); new identifiers can be created. Identifiers must be unique.
Python ✓ keywords, ✓ primitive elements, ✓ identifiers

3 important questions you should ask about any programming language:

- what are the primitive elements?
- how can we combine elements?
- how can we create our own elements?
many Python primitive elements – objects – are available when Python starts
many Python primitive elements – objects – are available when Python starts

```python
>>> 4  # objects have values
4  # literals evaluate to themselves
```
many Python primitive elements – objects – are available when Python starts

```python
>>> 4  # objects have values
4  # literals evaluate to themselves

>>> 'hello'
'hello'
```
many Python primitive elements – objects – are available when Python starts

```python
>>> 4  # objects have values
4  # literals evaluate to themselves

>>> 'hello'
'hello'

>>> len  # len is a built-in function
<built-in function len>
```
many Python primitive elements – objects – are available when Python starts

```python
>>> type ('hello')  # objects have types
<class 'str'>

>>> type (4)
<class 'int'>

>>> type(len)  # len is a built-in function
<class 'builtin_function_or_method'>
```
Python elements can be combined - expressions

```python
>>> 4 + 3  # Python Shell/interpreter
7  is a REPL.
```

```python
>>> len('hello')
5  expressions are executed
```

```python
>>> max(1, 2, 3)  # (evaluated) and
3  return a value
```

```python
>>>  # (which is printed when
code is executed in Shell)
```
Python elements can be combined - expressions

```python
>>> 4 + 3  # Python Shell/interpreter
7
>>> len('hello')
5 expressions are executed
>>> max(1, 2, 3) (evaluated) and
3 return a value
>>> (which is printed when code is executed in Shell)
```
Python elements can be combined - expressions

>>> type(len('hi'))

What gets printed?
elements can be named – assignment

```python
>>> my_identifier = 4  #Python assignment
```

**compare to**

```python
>>> my_identifier  #Python expression
4  #returns a value
```

Python assignment statements are not expressions and do not return a value.
elements can be named - assignment

```python
>>> my_identifier = 4  #Python assignment
```

```python
compare to
```

```python
>>> my_identifier  #Python expression
4  #returns a value
```

Python assignment statements are not expressions and do not return a value.

They associate an identifier (name) with a value (object); this information is stored in a Python namespace.
Python organizes and keeps track of identifiers in namespaces.
Python organizes and keeps track of identifiers in namespaces.

When Python starts up, two namespaces are available:

__builtins__ - Python built-in objects

__main__ - objects defined during Python session
>>> my_identifier = 4

Brand New Python Shell
>>> my_identifier = 4

>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'my_identifier']
>>> dir(__builtins__)
[,...,'False',...,'NameError','None',...,,'True',
'TypeError',...,,'abs','all','any','ascii','bin','bool',
'bytearray','bytes','callable','chr','classmethod',
'compile','complex','copyright','credits','delattr','dict',
'dir','divmod','enumerate','eval','exec','exit','filter',
'float','format','frozenset','getattr','globals','hasattr',
'hash','help','hex','id','input','int','isinstance',
'issubclass','iter','len','license','list','locals','map',
'max','memoryview','min','next','object','oct','open',
'ord','pow','print','property','quit','range','repr',
'reversed','round','set','setattr','slice','sorted',
'staticmethod','str','sum','super','tuple','type','vars',
'zip']
how can we extend the language – create our own elements?
- user-defined functions
- user-defined data types (CIS 211)
how can we extend the language – create our own elements?

- user-defined functions
- user-defined data types (CIS 211)

Recall: Python functions name an operation, e.g., built-in function abs
how can we extend the language – create our own elements?
- user-defined functions
- user-defined data types (CIS 211)

>>> abs

<built-in function abs> ...
how can we extend the language – create our own elements?

- user-defined functions
- user-defined data types (CIS 211)

```python
>>> help(abs)
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.
```
how can we extend the language – create our own elements?

- user-defined functions
- user-defined data types (CIS 211)

```python
>>> abs()  # functions are callable
```

??
how can we extend the language – create our own elements?

```python
>>> abs()
TypeError: abs() takes exactly one argument (0 given)
```

```python
>>> abs(7)  # function calls are expressions
7  # they evaluate to a value
# (another Python object)
```
Python Functions

The general form of a function call is:

<function name> (<arguments>)
Python Functions

The general form of a function call is:

<function name>(<arguments>)

```python
>>> abs(-7)
7
>>> abs(99)
99
```
Python Functions

The general form of a function call is:

<function name> (<arguments>)

```python
>>> pow(3, 2)
9
>>> 3 ** 2
9
```
Python Functions

The general form of a function call is:

<function name> (<arguments>)

```python
>>> pow(3, 2)
9
```

```python
>>> 3 ** 2
9
```

“Syntactic Sugar”
how can we extend the language – create our own elements?

– user-defined functions

For example

def twice(): # header incl. parameters
    """ # docstring
    ""

    result = 3 * 2 # code
    return result # value to return
For example

def twice():  # header incl. parameters
    """  # docstring
    """

    result = 3 * 2  # code

    return result  # value to return

Defining a function is like defining a variable – the function name/identifier refers to the operations specified in the function definition.
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'twice']

>>> twice
<function twice at 0x100560e18>

>>>
def twice():
    """ docstring goes here
    """
    result = 3 * 2
    return result

Functions are "callable", or "executable", data types – when a function is called, the operations specified in the function definition are executed.
def twice():
    """ docstring goes here
    """
    result = 3 * 2
    return result

Calling/executing a user-defined function is the same as calling/executing a built-in function:

>>> twice()
6
def twice(n):  # n is a parameter
    """ docstring goes here """

    result = n * 2
    return result

>>> twice(3)  # 3 is an argument
6

>>> twice(5)  # 5 is an argument
10
def twice(n):
    """ (number) -> number type contract - specify type for each parameter and type of returned value
    Returns double the brief description - incl. value of input number n. returned value
    """

>>> twice(3) examples of use – always
6 include at least two
>>> twice(10.5)
21.0
""

result = n * 2
return result
>>> help(twice)

(number) -> number

Returns double the value of input number x.

>>> twice(3)
6
>>> twice(10.5)
21.0