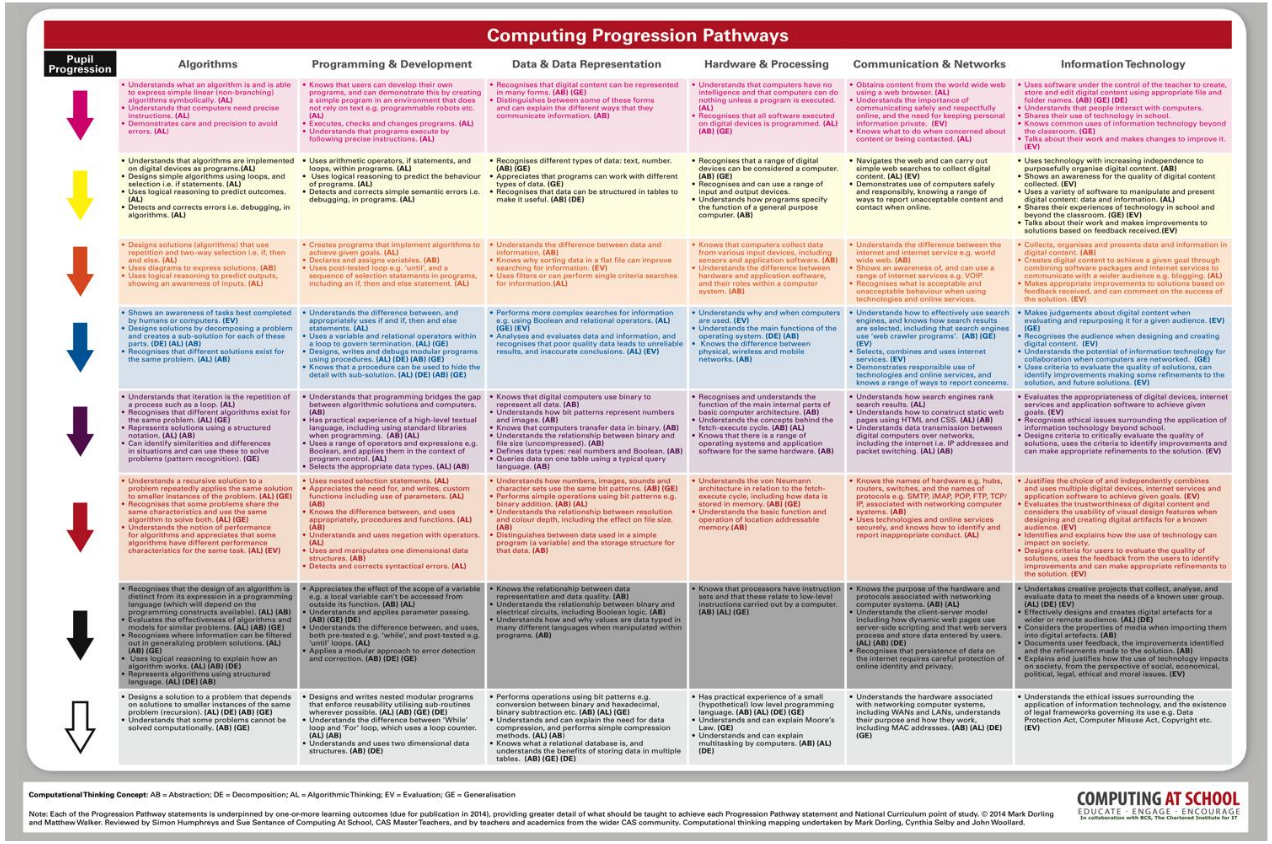
Supporting the transition from block to text based programming languages

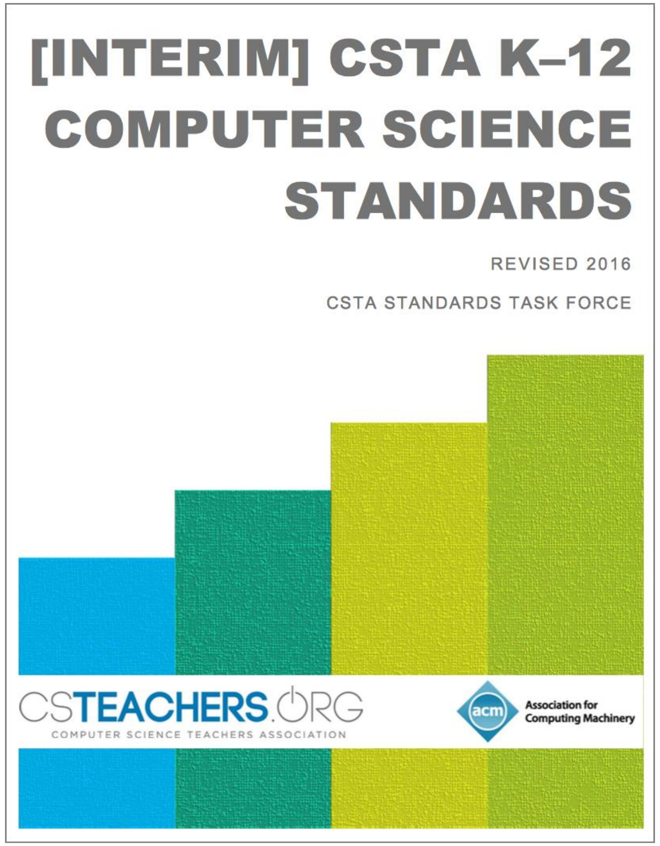
By Andrew Csizmadia, Mark Dorling & Steve Bunce

CONSTRUIT 2017
Warwick University

What do we have to teach?

Programmes of Study
(Concepts and key constructs)





[Document source](#) [Document source](#)

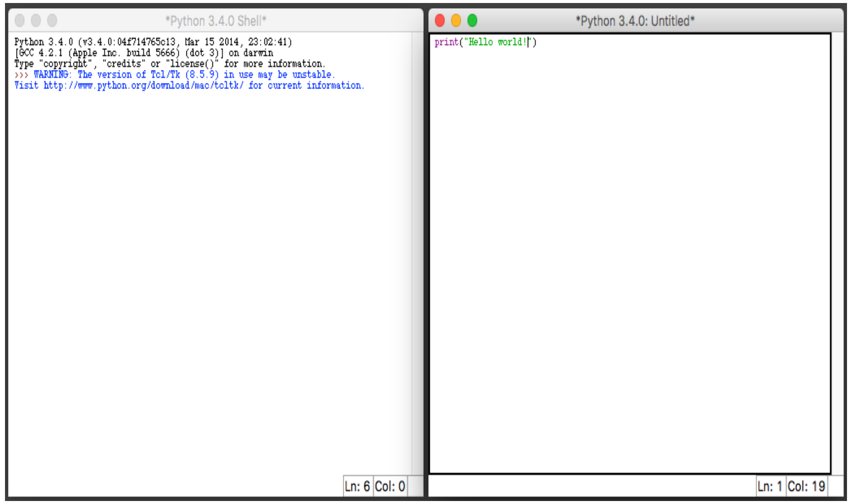
At Key Stage 4

CAS Category	Topic	AQA 16	AQA 16 Learning Objective	OCR 16	OCR 16 Learning Objective	Edexcel 16	Edexcel 16 Learning Objective	Eduqas 16	Eduqas 16 Learning Objective	Eduqas 16	IGCSE 16 Learning Objective
Algorithms	Representing algorithms	3.1.1.1	Understand and explain the term algorithm.	2.1.1	computational thinking: * abstraction * decomposition * algorithmic thinking	1.1.1	understand what an algorithm is, what algorithms are used for and be able to interpret algorithms (flowcharts, pseudo-code, written descriptions, program code)				
		3.1.1.2	Understand and explain the term decomposition.								
		3.1.1.3	Understand and explain the term abstraction.								
						1.2.2	be able to decompose a problem into smaller sub-problems				
						1.2.3	understand how abstraction can be used effectively to model aspects of the real world				
						4.4.3	understand how software can be used to simulate and model aspects of the real world	2.1.2	Use abstraction effectively to model selected aspects of the external world in an algorithm or program.		
								2.1.3	Use abstraction effectively to appropriately structure programs into modular parts with clear, well- documented interfaces.		
		3.1.1.4	Use a systematic approach to problem solving and algorithm creation representing those algorithms using pseudo-code and flowcharts.	2.1.4	how to produce algorithms using: * pseudocode * using flow diagrams	1.1.2	understand how to create an algorithm to solve a particular problem, making use of programming constructs (sequence, selection, iteration) and using appropriate conventions (flowchart, pseudo-code, written description, draft program code)	2.2.1	Use common methods of defining algorithms, including pseudo-code and flowcharts.	2.1.2.5	understand and use standard flowchart symbols to represent the above statements, commands and structures
								2.1.1	Use a systematic approach to problem solving including the use of decomposition and abstraction.	2.1.1.9	produce an algorithm for a given problem (either in the form of pseudocode or flowchart)
										2.1.2.1	understand and use pseudocode for assignment
										2.1.2.2	understand and use pseudocode, using the following conditional statements
										2.1.2.3	understand and use pseudocode, using the following loop structures
										2.1.2.4	understand and use pseudocode, using the following commands and statements
		3.1.1.5	Explain simple algorithms in terms of their inputs, processing and outputs.			4.1.1	understand the input-process-output model				
		3.1.1.6	Determine the purpose of simple algorithms	2.1.5	interpret, correct or complete algorithms.	1.1.3	understand the purpose of a given algorithm and how an algorithm works			2.1.1.3	work out the purpose of a given algorithm
						1.2.1	be able to analyse a problem, investigate requirements (inputs, outputs, processing, initialisation) and design solutions				
	Efficiency of algorithms	3.1.2.1	Understand that more than one algorithm can be used to solve the same problem.								
						1.1.4	understand how to determine the correct output of an algorithm for a given set of data				
				2.1.5	interpret, correct or complete algorithms.	1.1.5	understand how to identify and correct errors in algorithms			2.1.1.8	identify errors in given algorithms and suggest ways of removing these errors
						1.1.9	be able to evaluate the fitness for purpose of algorithms in meeting specified requirements efficiently using logical	2.2.16	Explain how an algorithm or program works and evaluate its fitness for purpose in meeting requirements.	2.1.1.10	comment on the effectiveness of a given solution
		3.1.2.2	Compare the efficiency of algorithms explaining how some algorithms are more efficient than others in solving the same problem.					2.2.17	Evaluate the efficiency of an algorithm or program using logical reasoning and test data.		
						1.1.7	understand how the choice of algorithm is influenced by the data structures and data values that need to be manipulated				
	Searching algorithms	3.1.3.1	Understand and explain how the linear search algorithm works.	2.1.2	standard searching algorithms: * binary search * linear search	1.1.8	understand how standard algorithms (bubble sort, merge sort, linear search, binary search) work	2.2.15	Explain and use linear and binary search algorithms.		
		3.1.3.2	Understand and explain how the binary search algorithm works.								
		3.1.3.3	Compare and contrast linear and binary search algorithms.								
	Sorting algorithms	3.1.4.1	Understand and explain how the merge sort algorithm works.	2.1.3	standard sorting algorithms: * bubble sort * merge sort * insertion sort	1.1.8	understand how standard algorithms (bubble sort, merge sort, linear search, binary search) work	2.2.14	Describe the characteristics of merge sort and bubble sort algorithms.		
		3.1.4.2	Understand and explain how the bubble sort algorithm works.								
		3.1.4.3	Compare and contrast merge sort and bubble sort algorithms.								

Supporting the transition from: Block to Text

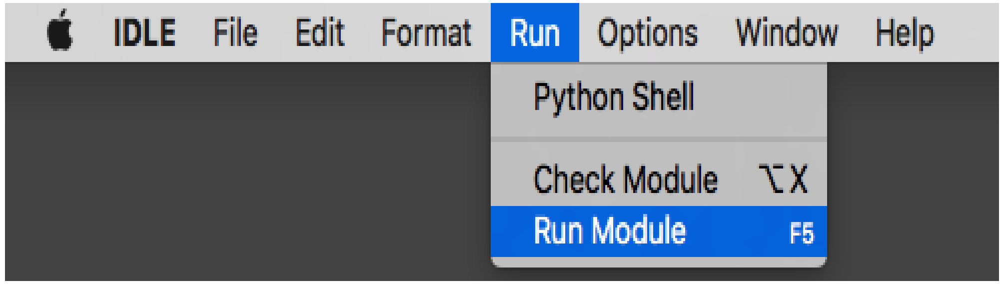
IDLE: Script mode

- Interactive mode is great but its not designed to create programs that you can save and run later.
- The Script mode enables you to write, save, open and edit programs.
- File > New File

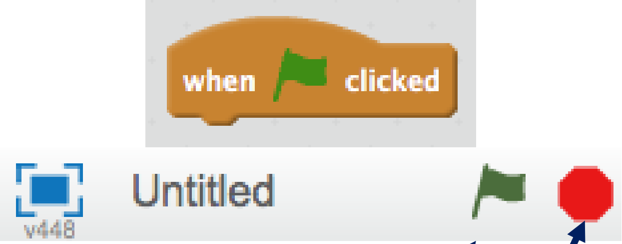


IDLE: Script mode

- Run > Run Module



- Save the file.

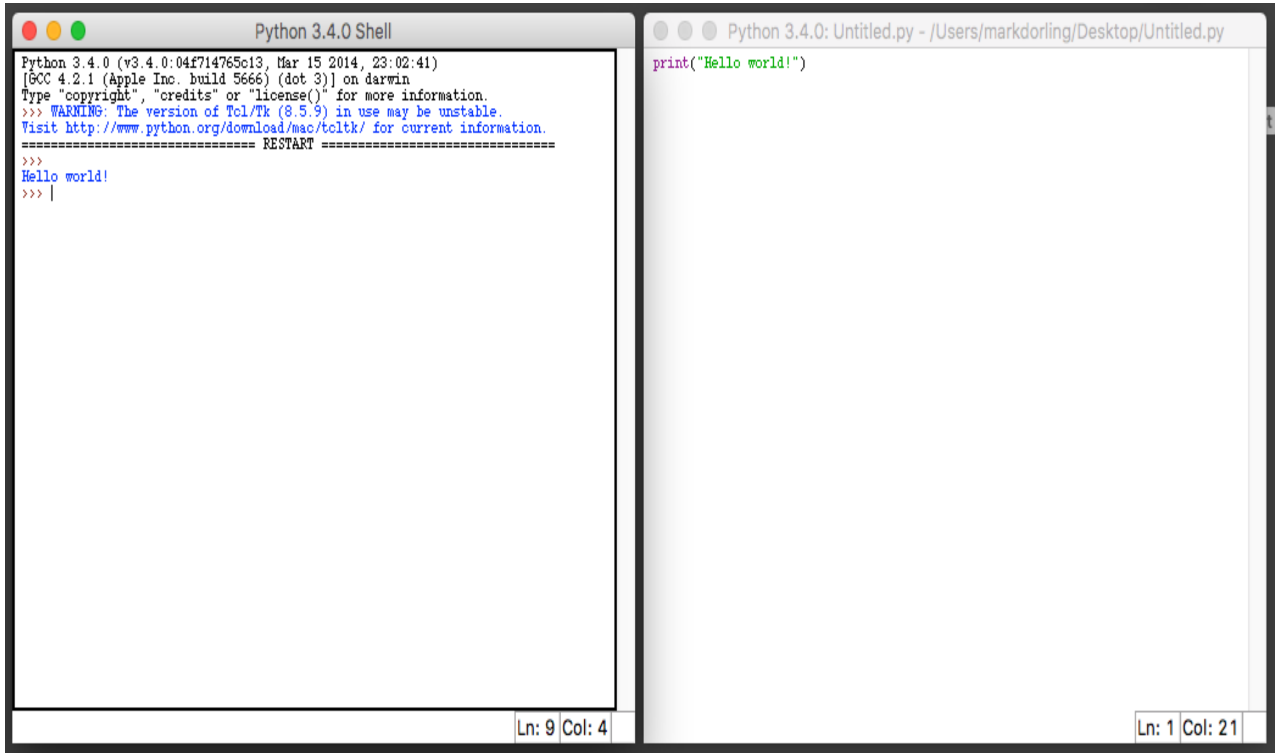


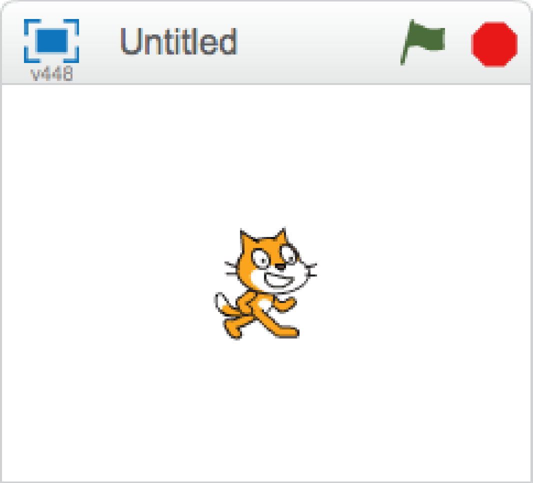
Start the program

Kill the program

IDLE: Script mode

- See the results!

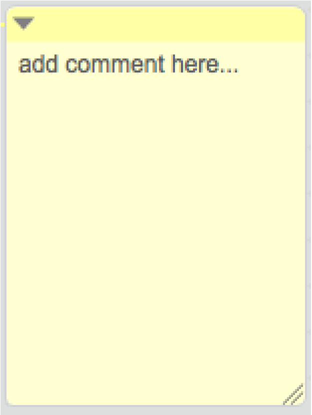




Using comments

- It is good practice to use comments at the of a program to give:
  - Explanation of what it does
  - The author
  - The version:
  - The date:
  - Any copyright information

```
#####
#
# Explanation of what the program does
#
# Author :
# Version :
# Date:
####
```



Wait in Python

- The is no print for ‘n’ seconds in Python like in Scratch and Snap!

- How do we get around this?
- **Python's time module has a handy function called sleep()**

Wait (sleep) in Python

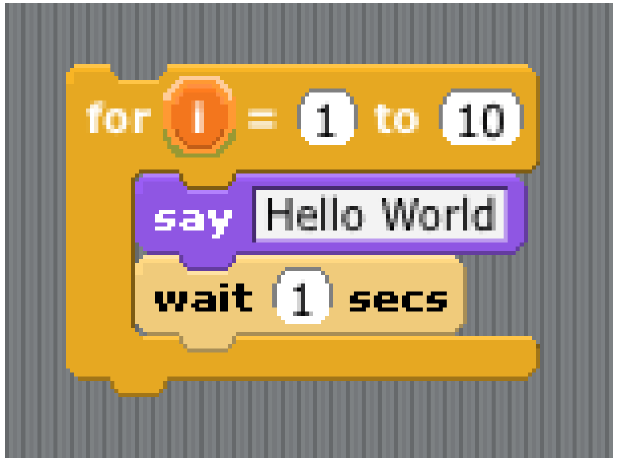
- The syntax is this: time.sleep(secs)
- We need to remember to import the time module.

```
import time

for i in range (1,10):
    print("Hello World!")
    time.sleep(1)
```

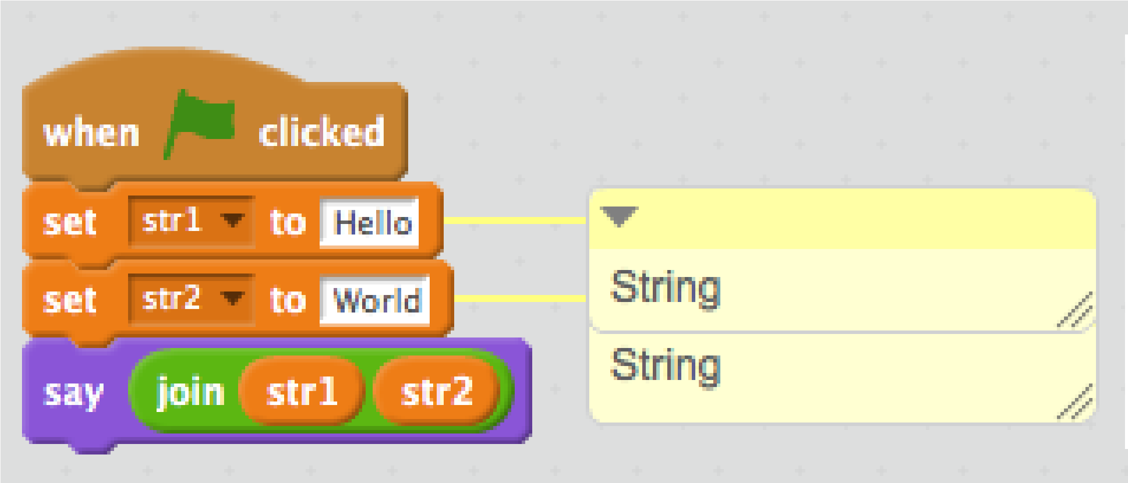
python

SCRATCH



Snap!

Introducing: Data Types

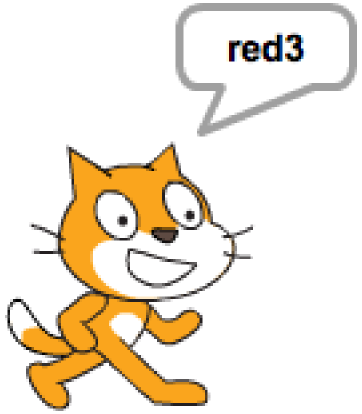



```python
str1 = str("Hello")
str2 = str("World!")

print(str1 + str2)
```

Introducing: Concatenation

print “red” + 3



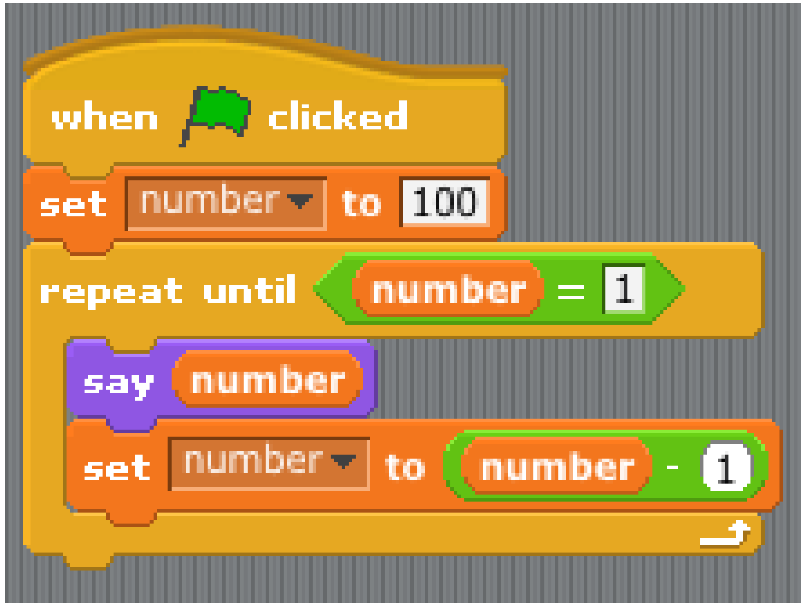
Traceback (most recent call last): File "", line 1, in

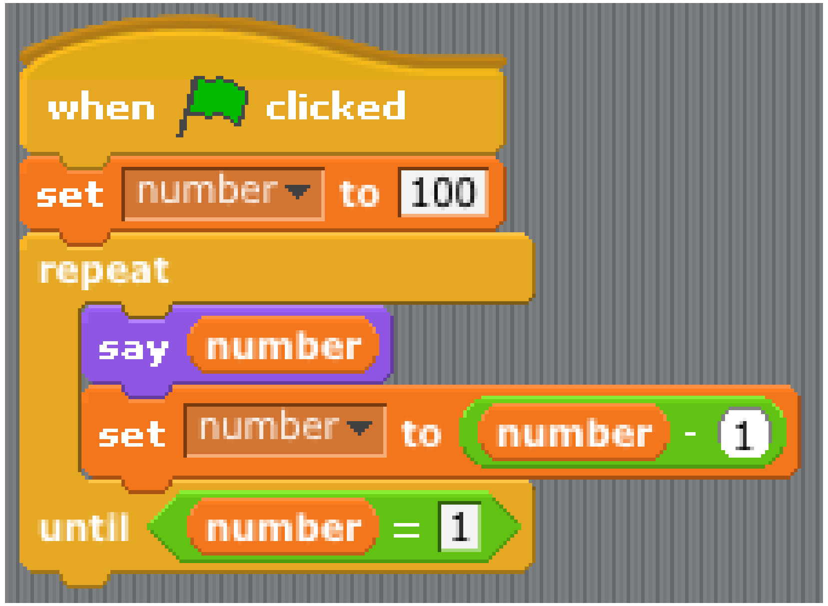
TypeError: cannot concatenate 'str' and 'int' objects

- Python doesn't know how to add a word and a number, so it says "cannot concatenate 'str' and 'int' objects.
- You cant add a str and a int togther. But you can turn an integer into a string if you use the str() function.

print “red” + str(3)

Bottom/post-tested loops



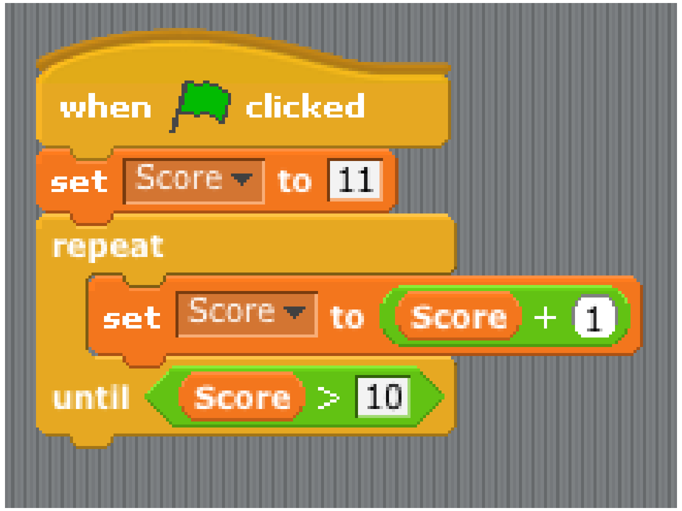


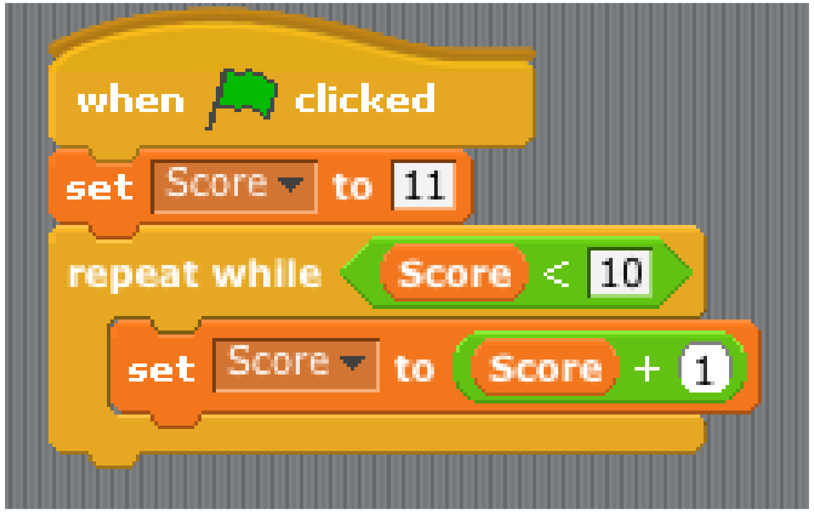
What if the count variable is set to 1 at the start of the program and not 100?

Introducing: Top & bottom tested loops

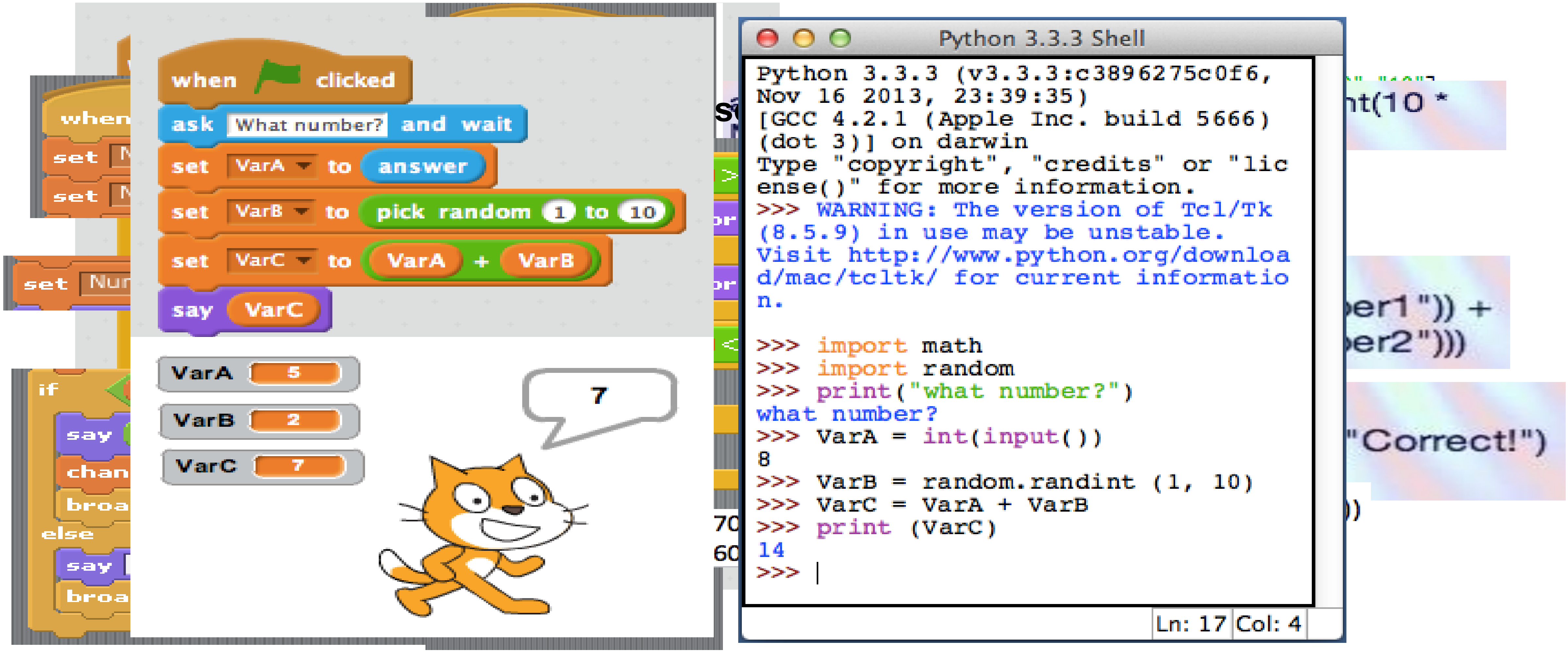
Repeat
print("Enter name of item")
input(item)
print("Another item?")
input(response)
Until response = 'N' or "No"

Repeat While response = 'N' or "No"
print("Enter name of item")
input(item)
print("Another item?")
input(response)





Range of languages



Range of problems

- Range of recursive and iterative activities
  - 10 green bottles
  - One man went to Mow
  - Fibonacci and Lucas series
- Rock Paper Scissors
- Hang man style games
- Searching Algorithms
  - Linear
  - Binary
- Sorting algorithms
  - Bubble
  - Insert
  - Merge
  - Selection

Transition…

KS3/4 python

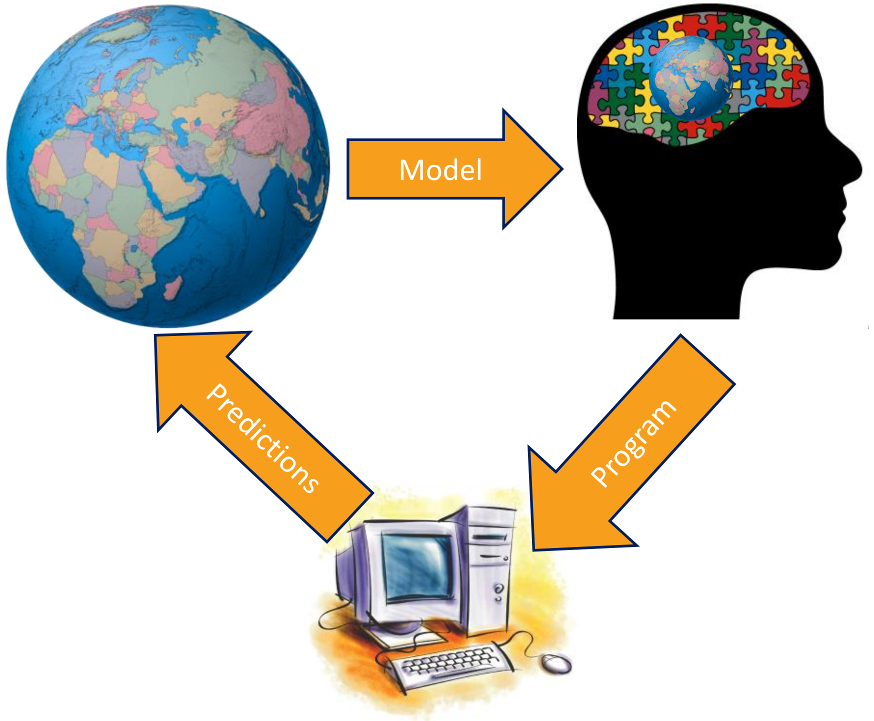
KS3 Snap!

KS2 SCRATCH

Exploring programming pedagogy

Why programming?

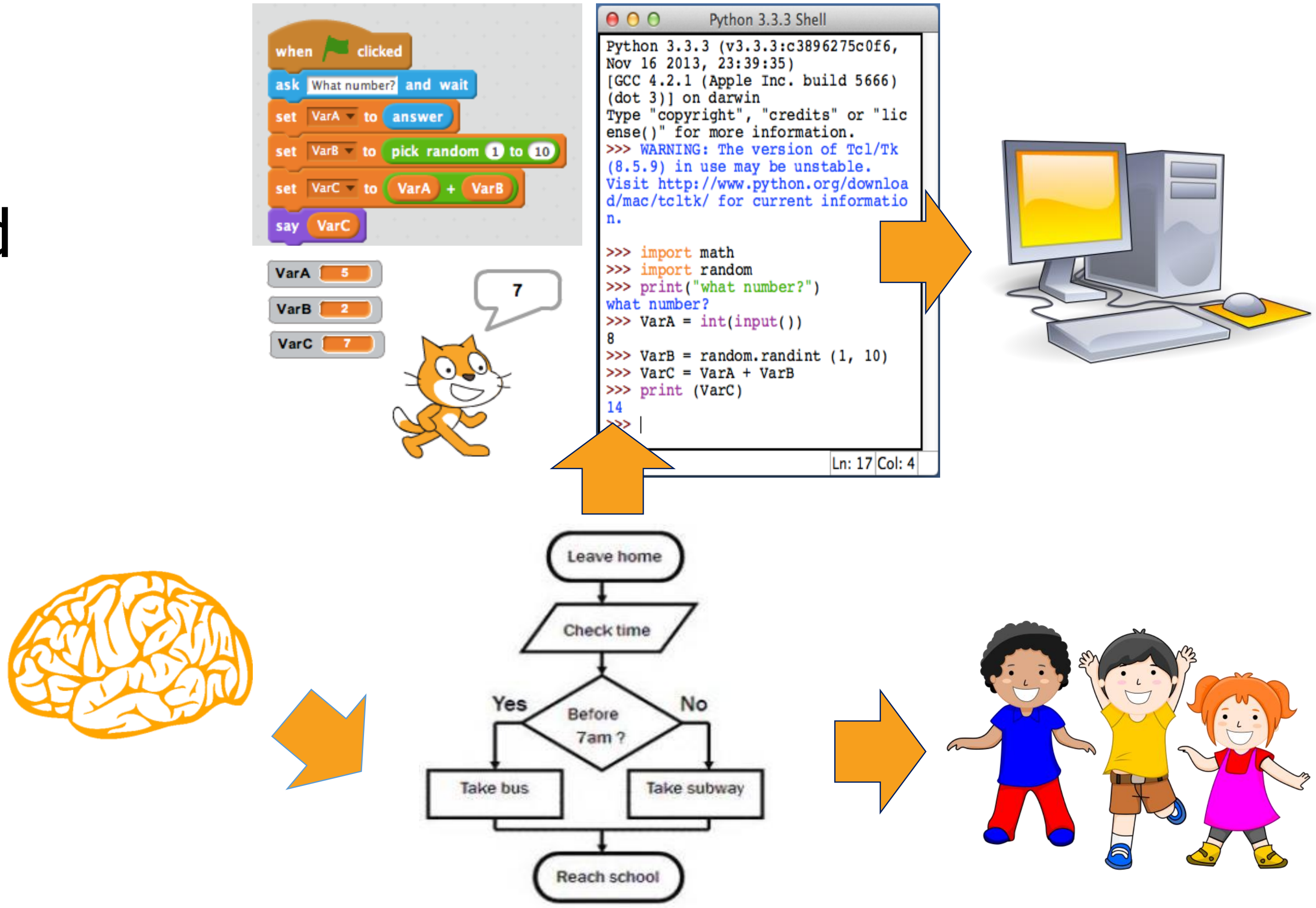
- We want to make models of the world to:
  - Understand it
  - Ask "what if" questions and predict the way it will change
- How do we make models?
  - Solving problems
    - Asking good questions
    - By characterising a problem
      - Looking for similar problems you already know how to solve
      - Think about what makes a problem similar to another



(Professor Greg Michaelson, Heriot-Watt University)

Abstractions…

- How do we turn models into programs?
  - Writing programs based on algorithms
  - Programming bridges gaps between thinking and computers?
  - Choosing appropriate technology



Our aim for learners is to…

- The McCracken group focused on problem solving in programming and suggested a 5-step process that students should learn:
    1. Abstract the problem from its description **(Abstraction)**
    2. Generate sub-problems **(Decomposition)**
    3. Transform sub-problems into sub-solutions **(Generalisation and Algorithmic Thinking)**
    4. Re-compose, and
    5. Evaluate and iterate **(Evaluation)**

McCracken et al. (2001)

Challenges for novice programmers

- Students might lack skills that are a precursor to problem-solving.

Lister et al. (2004)

- Being able to read and trace code is really important pre-cursor to the problem-solving needed to write code.

Lister et al. (2008)

- Novice programmers need to be able to trace code with greater than 50% accuracy before they can independently begin to write programs.

Lister (2011)

Essential Programming Skills

Create		**Design:** *Devise a solution structure* **Apply:** *Use a solution as a component in a problem*	**Model:** *Illustrate or implement an abstraction of a problem*	**Refactor:** *Redesign a solution for optimisation*
Apply	**Implement:** *put a completed design into code*	**Adapt:** *modify a solution for other domains* **Translate:**	**Debug:** *Both detect and correct flaws in design*	
-	**Recognise:** *Base knowledge and vocabulary for the domain*	**Trace:** *Desk check a solution*	**Present:** *Explain a solution to others* **Analyse:** probe the [time] complexity of a solution	**Relate:** *Understand solution in context of others*
	Know	**Understand**	**Analyse**	**Evaluate**

Fuller et al. (2007)

Use - Modify - Create

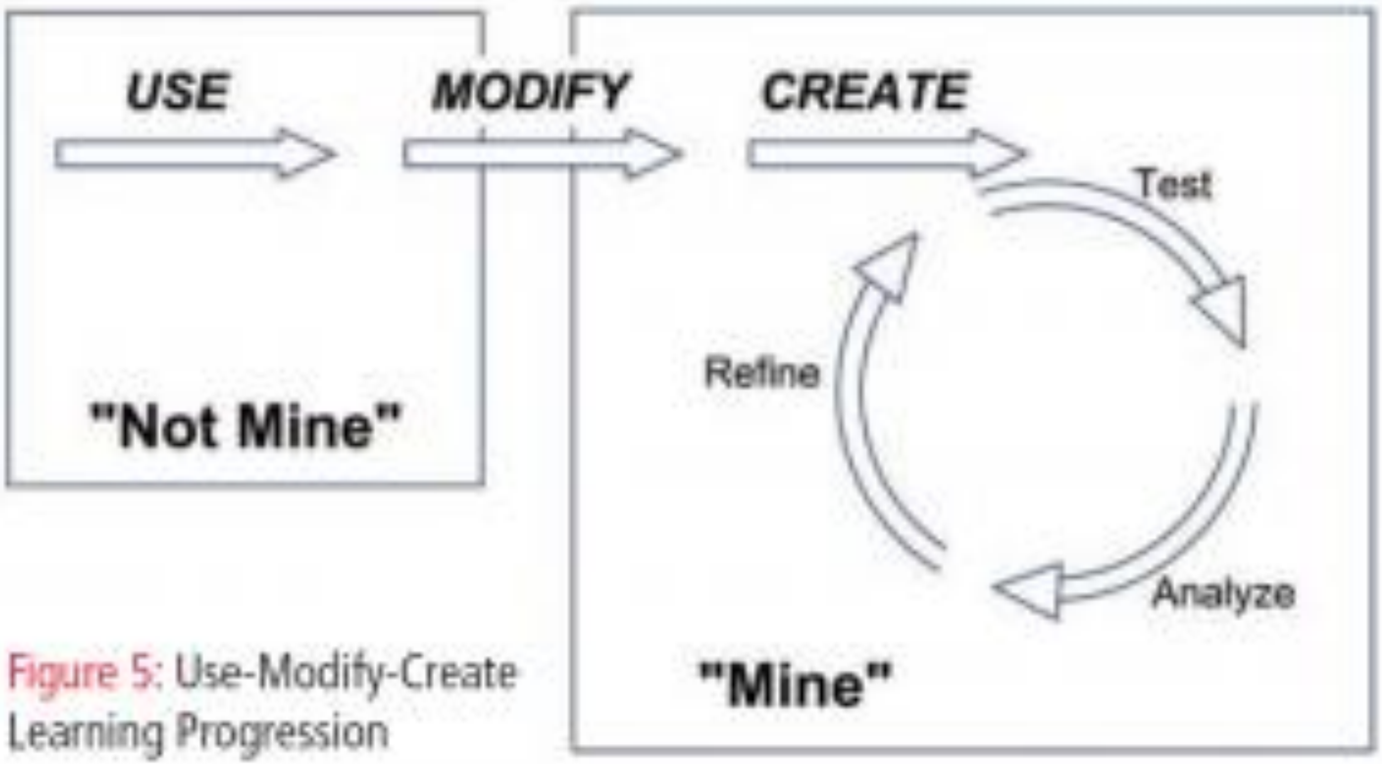
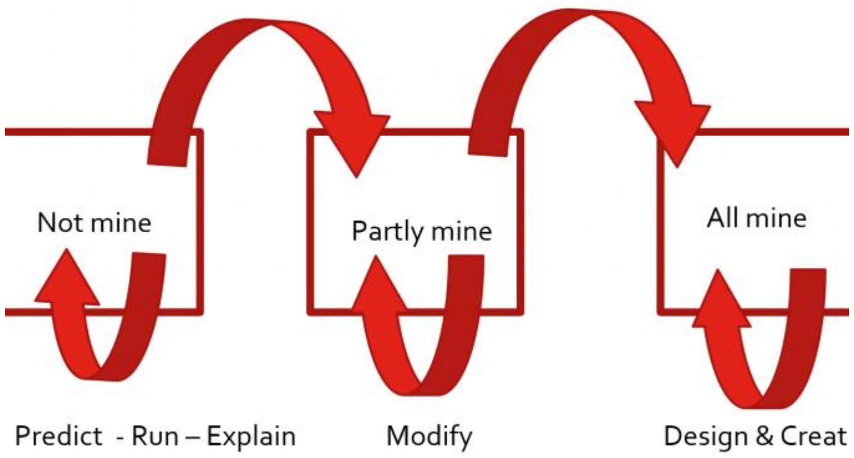


Figure 5: Use-Modify-Create Learning Progression

Lee et al. (2011)

Use - Modify - Create



- **Predict** – given a working program, what do you think it will do? (at a high level of abstraction)
- **Run** – run it and test your prediction
- **Explain/Articulate** – What does each line of code mean? (low level of abstraction). I'm not sure that explain is quite the right term.
- **Modify** – edit the program to make it do different things (high and low levels of abstraction)
- **Create/Design** – design/create is a key computational thinking skill.

Sentence. (2017)

Pulling this together

	Lee et al. (2011)	Fuller et al. (2007)	Sentence (2017)
CONSTRUCT	Novice coders constructing an understanding by using and running code provided	Construct knowledge: new vocabulary and knowledge about construct, reading and tracing, explaining, analysing and evaluating code	The code is 'not theirs'
CHANGE	Novice coders begin modifying and adapting code provided	Applying knowledge: Implementing code from a given design, adapting code for a different purpose, finding and correcting errors in code (debugging).	Making the code 'partly' thiers'
CREATE	Novice coders begin designing and writing their own code	reusing solutions in bigger solutions, designing and coding solutions, fixing smelly code (commenting and optimising code).	The code is 'fully theirs'

Historical: Choice of pedagogy

Constructivism: The learner is not a passive recipient of knowledge but that knowledge is 'constructed' by the learner.

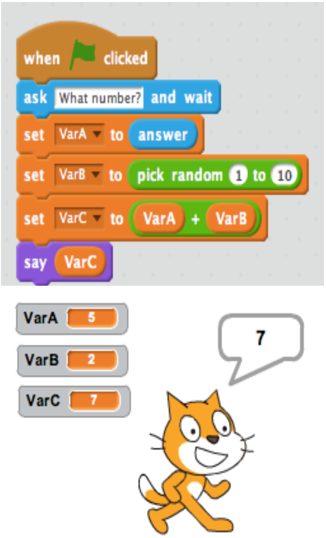
CS UNPLUGGED

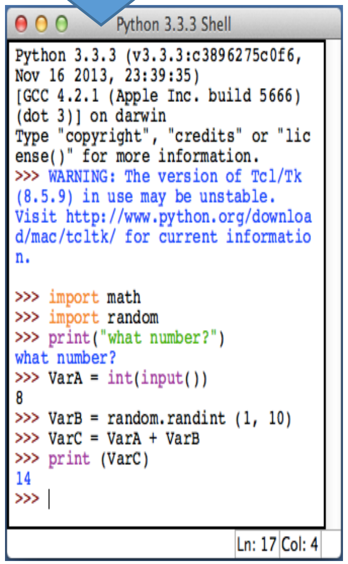
Computer Science without a computer

The Magic of Computer Science
www.cs4fn.org/magic/

Social Constructivism: Groups construct knowledge for one another, collaboratively creating a smaller culture of shared artifacts with shared meaning.

Constructionism: The idea that learners' learn best through building things that are tangible and sharable with the public





Example pedagogies

- **Code walkthroughs:** Learners step through code predicting outputs
- **Collaborating on solutions:** Writing algorithms and code in groups
- **Scaffolding:** Insert comments in into existing code
- **Code debugging:** Finding errors in given code e.g. spot the difference
- **Flipped learning:** Class time to collaborate and compare solutions

Van Gorp and Grissom (2001)

Supporting constructivist

- Using examples that are **relevant to students' own experiences** e.g. relating to real-world experiences
- **Active learning experiences** e.g. unplugged, kinesthetic activities
- **Learning by exploration** e.g. exploring programming environments and open-ended tasks
- **Learning by solving problems** e.g. self-directed projects and problem-solving
- **Open-ended discussion and working in groups** e.g. paired and group problem-solving.

Supporting social constructivist

- Software developers generally spend:
  - **30%** of their time working alone
  - **50%** of their time working with one other person
  - **20%** of their time working with two or more people

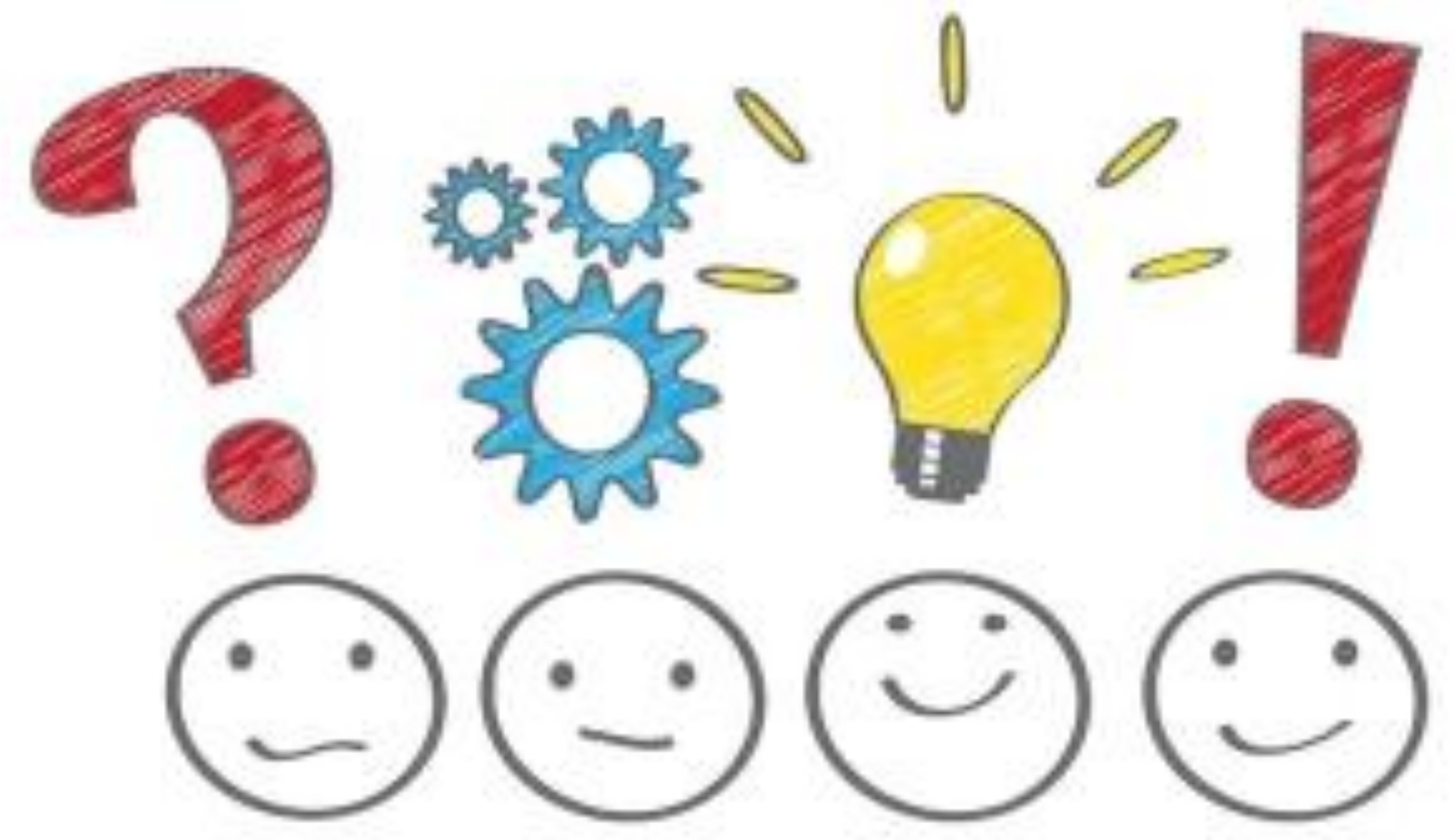
DeMarco and Lister (1987)

Three forms of peer-based interaction in the classroom:

1. Tutoring, where the less capable are guided by the more capable;
2. Co-operation, where learners work on different parts of the task;
3. Collaboration, where learners work jointly on almost all parts of the task.

Jehng (1997)

Thanks for listening!



References

- AQA GCSE Computer Science: http://www.aqa.org.uk/subjects/computer-science-and-it/gcse/computer-science-8520
- Cambridge iGCSE Computer Science: http://www.cie.org.uk/programmes-and-qualifications/cambridge-igcse-computer-science-0478/.
- Csizmadia, A., Cuzon, P., Dorling, M., Humphreys, S., Ng, T., Selby, C. and Woollard, J. (2015). *Computational thinking: A guide for teachers*: https://community.computingatschool.org.uk/resources/2324
- Computing At School (2015). *Lesson observation form with prompts*: http://www.quickstartcomputing.org/secondary/section4.html
- Department for Education (2014). *Computing Programmes of Study*: https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study

References

- DeMarco, T., and Lister, T. (1987). *Peopleware*. New York: Dorset House Publishers.
- Dorling, M. & Stephens, S. (2016). *Problem solving and Computational Thinking rubric*: http://community.computingatschool.org.uk/resources/4793
- Dorling, M. & Walker, M. (2014), *Computing At School Progression Pathways*: https://community.computingatschool.org.uk/resources/1692
- Edexcel GCSE Computer Science: http://qualifications.pearson.com/en/qualifications/edexcel-gcses/computer-science-2016.html
- Fuller et. Al. (2007). *Developing a computer science-specific learning taxonomy*: http://dl.acm.org/citation.cfm?id=1345438

References

- Jehng, J. C. J. (1997). The psycho-social processes and cognitive effects of peer-based collaborative interactions with computers. *Journal of Educational Computing Research*, 17, 19-46.
- Hazzan, O., Lapidot, T. and Ragonis, N., (2015). *Guide to teaching computer science: An activity-based approach*. London: Springer.
- Lister, R. (2011). *Concrete and other neo-piagetian forms of reasoning in the novice programmer.*
- Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., . . . Thomas, L. (2004). *A multi-national study of reading and tracing skills in novice programmers*.
- Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., . . .Werner, L. (2011). Computational thinking for youth in practice. *ACM Inroads, 2,* 32–37.

References

- Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). *Relationships between reading, tracing and writing skills in introductory programming*.
- McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagen, D., Kolikant, Y., Laxer, C., Thomas, L., Utting, I., and Wilusz, T. (2001). A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-year CS Students. SIGCSE Bull., 33(4). pp 125-140.
- OCR GCSE Computer Science: http://www.ocr.org.uk/qualifications/gcse-computer-science-j276-from-2016/.
- Roth, W.-M. (1993). Construction sites: Science labs and classrooms. In K. Tobin (Ed.), The practice of constructivism in science education, (pp. 145-170). Hillsdale, NJ: Erlbaum.

References

- Sentance, S. and Csizmadia, A., (2015). Teachers' perspectives on successful strategies for teaching Computing in school
- Van Gorp, M. J., and Grissom, S. (2001). An empirical evaluation of using constructive classroom activities to teach introductory programming.
- Williams, Wiebe, Yang, Ferzli, and Miller, "In Support of Pair Programming in the Introductory Computer Science Course," Computer Science Education, vol. 12, pp. 197-212, 2002.
- WJEC GCSE Computer Science: http://wjec.co.uk/qualifications/computer-science/computer-science-gcse/